M6x/cM6x Development Package Manual

The M6x/cM6X Development Package Manual was prepared by the technical staff of Innovative Integration, February 2000.

For further assistance contact:

Innovative Integration
5785 Lindero Canyon Road
Westlake Village, California 91362

PH:(818) 865-6150
FAX:(818) 879-1770
email:techsprt@innovative-dsp.com
Website:www.innovative-dsp.com

VSS\M62\documents\Hw-Sw Manual\M62manual.book

CHAPTER 1 Introduction

This document describes the Zuma software development environment for Innovative Integration (I.I.) digital signal processor (DSP) cards. The environment comes complete with ANSI compliant C code Compilation, Assembler, Linking, Debugging, and Windows interface software and represents the most complete package available for DSP code creation for Texas Instruments DSP processors.

Each Developer's Package consists of four major features:

- TMS320-based DSP board
- Texas Instruments Floating Point C Compiler/Assembler toolset
- Code Composer JTAG-based hardware-assisted debugger
- Zuma software toolset including:

  DSP Peripheral Library - supporting on-board peripherals and DSP functions, with full source code

  Custom 32-bit Windows 95/98/NT compatible dynamic link library (DLL) - which utilizes a custom, 32-bit, Ring 0/Kernel-mode device driver for host PC software application development

  Host Support Applets - for automatic program download, terminal emulation, COFF file dumping and on-board flash programming

  Sample Applications - showing Host PC as well as target DSP coding techniques

This manual discusses installation issues and includes full documentation on all Innovative Intergration software tools (please see the accompanying manuals for specific information on the T.I. toolset or Code Composer Studio software packages). Installation is discussed first, followed by brief introductions to each of the software packages and instructions on their use. General software development issues are presented, and a tutorial on DSP software development, particularly as it relates to the integrated use of the software packages included in this kit, are also discussed. References are given for the peripheral libraries and host DLL packages in the Appendices.

A Note about this Manual

Certain typography conventions are used in this manual to indicate user operations, file types, etc., as follows:

- Windows application menu commands are identified and presented as pipe-delimited strings indicating the menu entries which are being discussed. For example, the Load Program menu item under the File menu in the Code Composer package would be named by the following string:

  File | Load Program

  Computer readable files and keyboard input/output are represented in Courier font, with user input in bold. For example, a program file will be referred to by name as

  C:\SBC32\TALKER\TALKER.OUT

  while user input and commands look like

  ROM MYPROG.OUT

CHAPTER 2

Installation

Installation of the Zuma toolset consists of both hardware and software installation procedures. This document contains complete installation directions for Innovative Integration's Development Package. This document also details the features of the Innovative Integration software generation tools, applets, utilities and peripheral library functions for the target DSP board. Refer to the *Hardware* section of this manual for a discussion of hardware-specific configuration information.

The Development Package consists of software elements developed by Innovative Integration, Texas Instruments, and other sub-vendors. This document is intended to augment, not replace, the *Installation Supplement* and the documentation provided with the TI C compiler, Code Composer Studio, and other third-party software packages. Refer to the documentation provided with those products for a complete discussion of their features and use.

Host Hardware Requirements

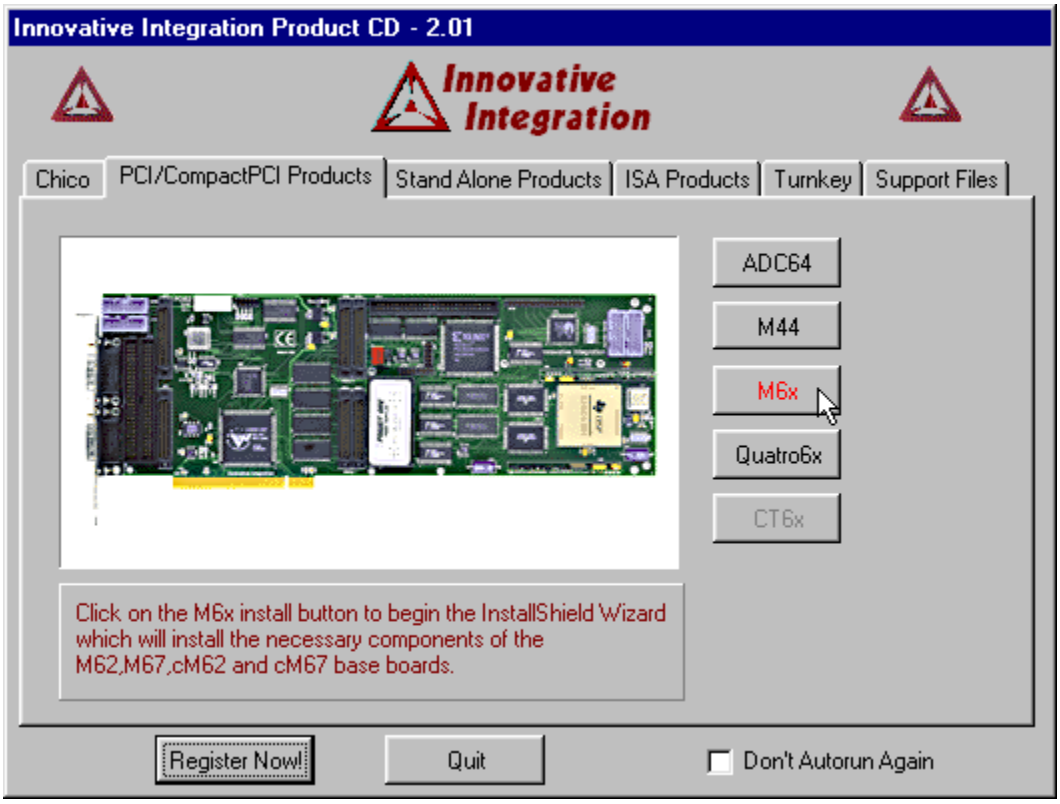
The software development tools for the Zuma toolset require an IBM or 100% compatible 486-class or higher machine for proper operation (Pentium-class machines are highly recommended). The host system must have at least 16 Mbytes of memory, up to 84 Mbytes available hard disk space, and a CDROM drive. Windows 95/98 or NT (referred to herein simply as *Windows*) is required to run the developer's package software, and is the target operating system for which host software development is supported.

Software Installation

The installation consists of the following major components: TI Compiler install, Code Composer Studio install, and the Zuma Toolset install. The installation time will take approximately 10 to 15 minutes depending on the system's speed. If you have not purchased some of the above listed components, you can go through the custom install to unselect the non purchased items.

Begin Installation

To install DSP board based products, the Logger PCI, or any of our support DLL's start the host operating system and insert the installation CD. If the CD does not auto start, click on the <**Start**> button, then <**Run**>. Enter the path to the SETUP.EXE program located at the root of your CD-ROM drive, i.e. D:\SETUP.EXE. The setup program will run. Select the tab for the type of installation you are going to do. From there, select the exact product you wish installed. All necessary components including the Hasp key drivers, the board drivers, the peripheral libraries and debugging drivers will be automatically installed to your host PC using Install Shield.

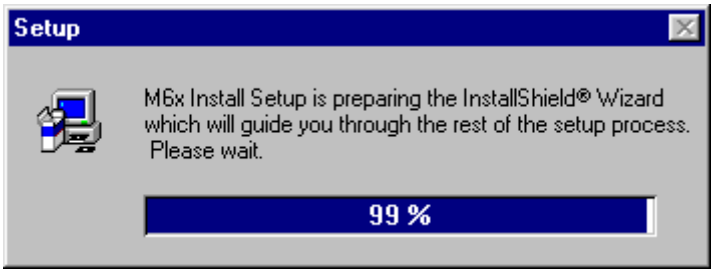


Important, Microsoft Windows NT Users Please Note: The installation of the NT Device Driver for a Peripheral Library requires the installing user to have Administrator rights on the system. This does not have to be the actual Administrator login, as long as the rights are the same.

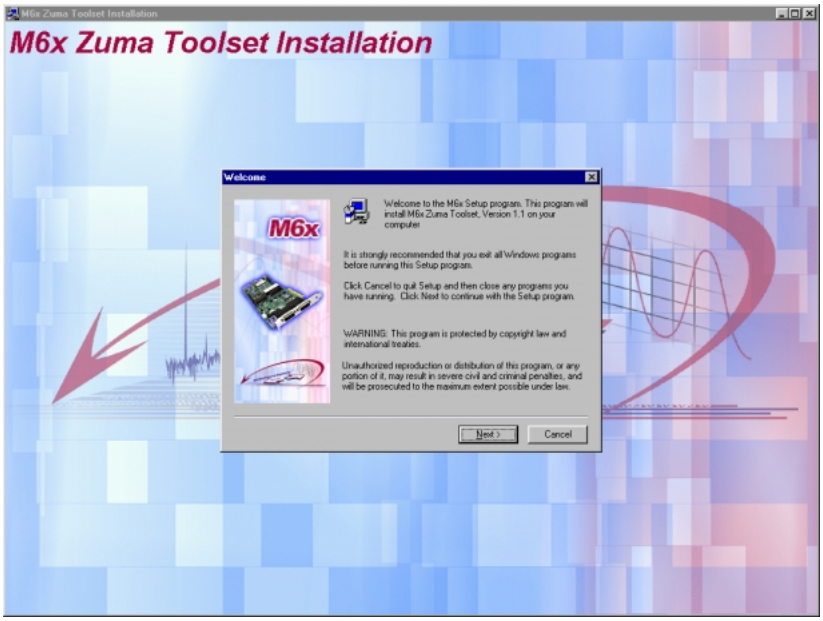
Additionally, applications that receive interrupts from a target board must be run by a user with Administrator rights.

Installation Instructions

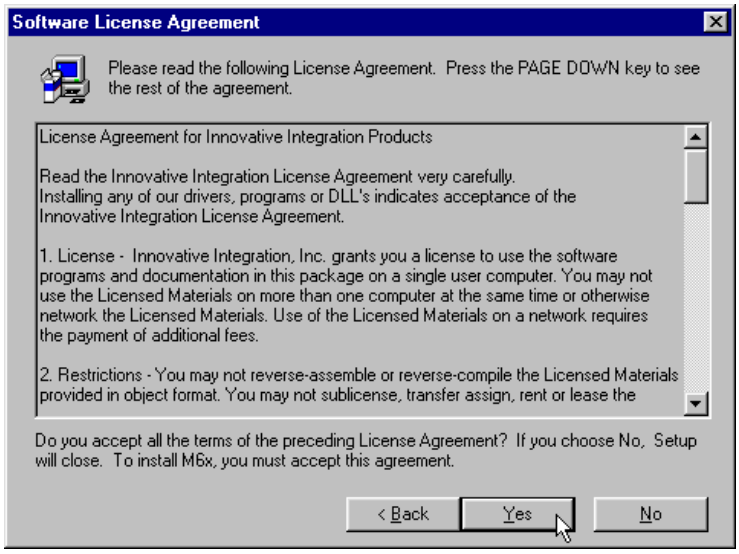
InstallShield will be automatically invoked. The following screen will be displayed while Install Shield is copying the setup files onto your system.



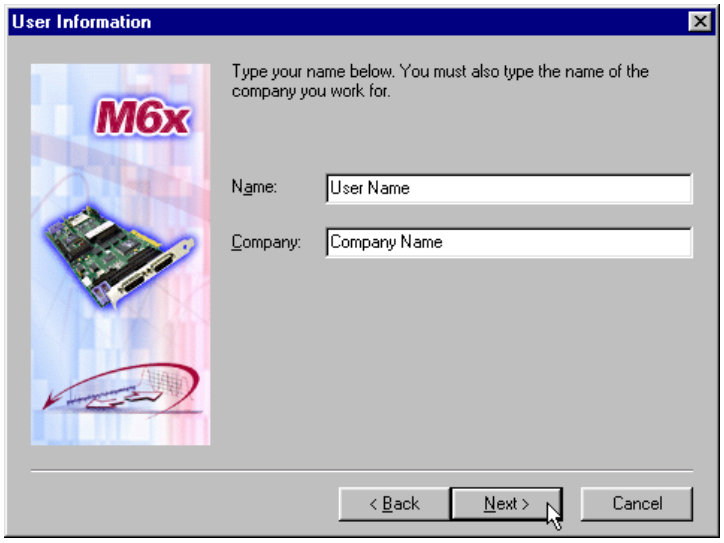
The first screen that will appear is the welcome screen. Click **<Next>**



Now, the license agreement dialog box is displayed. After reviewing it, if you agree, Click <**Yes**>. If you do not agree, contact the sales support department.



Next you will be prompted to enter your user information. Enter the User Name, press <**Tab**>, the Company Name and click <**Next**>

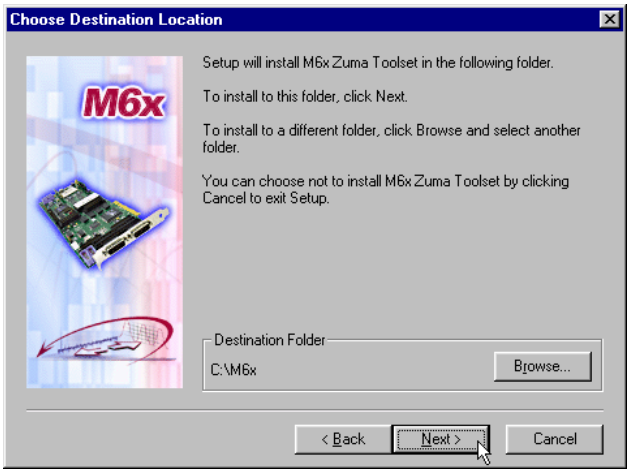


Note: Although any installation drive and path may be specified when installing the Peripheral Library, Innovative Integration highly recommends that the default installation drive and directory be used whenever possible. The Code Composer workspace files for the sample DSP applications have been

setup with the default directory paths in mind. If an alternate drive or directory is used, the workspace project setups will need to be changed to reflect the new path. See the Code Composer documentation for more details on the use of projects and workspaces.

The Innovative Integration Peripheral Library is included with the purchase of all Development Packages, and includes example DSP software and a complete set of peripheral control libraries as well as sample host applications and DLL's for use in host code development.

Click **<Next>** when you are done.



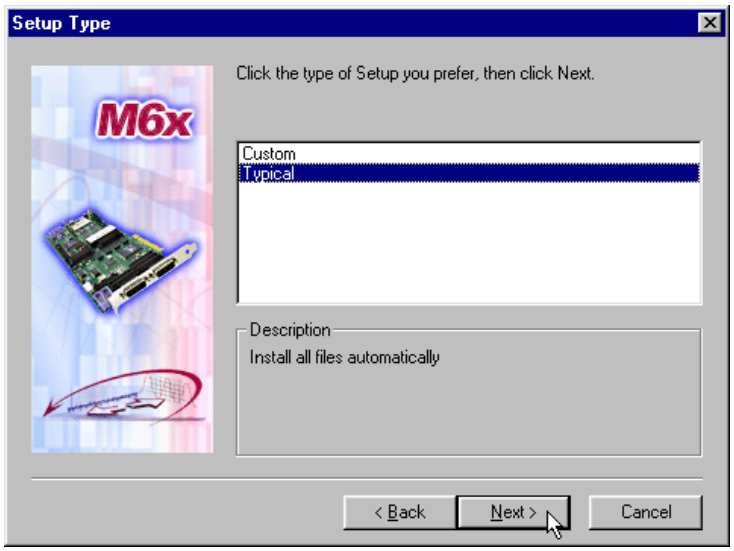
Now you may choose the installation type. If you choose "Typical", then the following components will be installed:

- Code Composer Studio
- Peripheral Libraries
- HASP Key drivers
- DSP board drivers
- JTAG debugger driver (see note below)
- PortIo (for NT installs only)

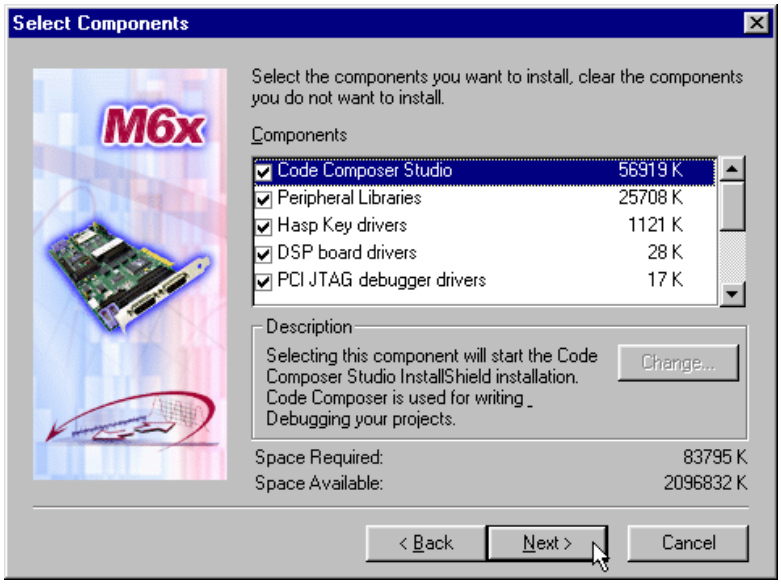
NOTE: Both PCI and ISA JTAG drivers have been provided on the installation CD. The PCI JTAG driver is the default. If you are using an ISA JTAG model, select custom install. Then de-select the PCI JTAG driver and select the ISA JTAG driver.

If you choose "Custom", then you may choose the components you would like installed

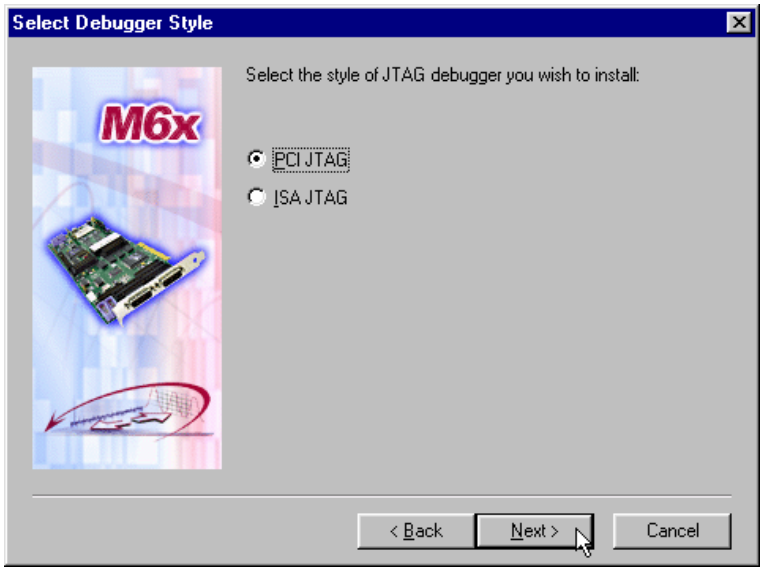
Make your selection and click <**Next**>.



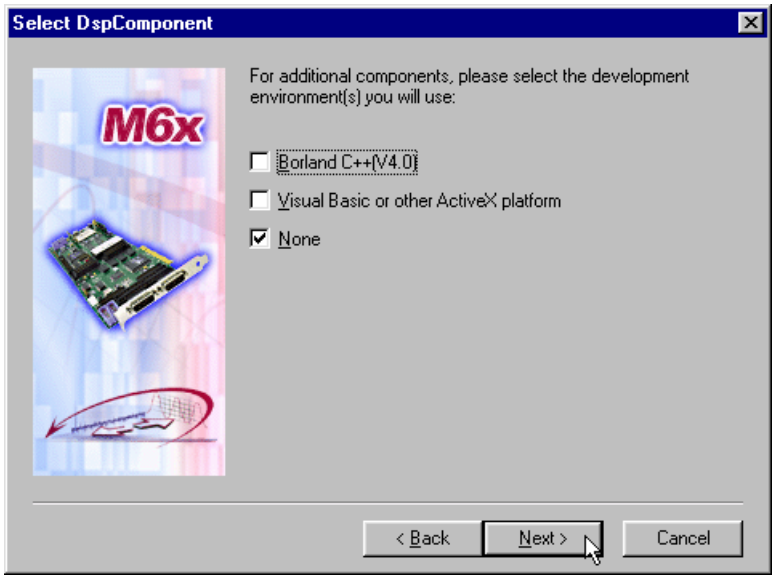
If you chose "Custom", then you will see the following screen in order to choose the components you want to install. Click <**Next**> when you have made your selections.



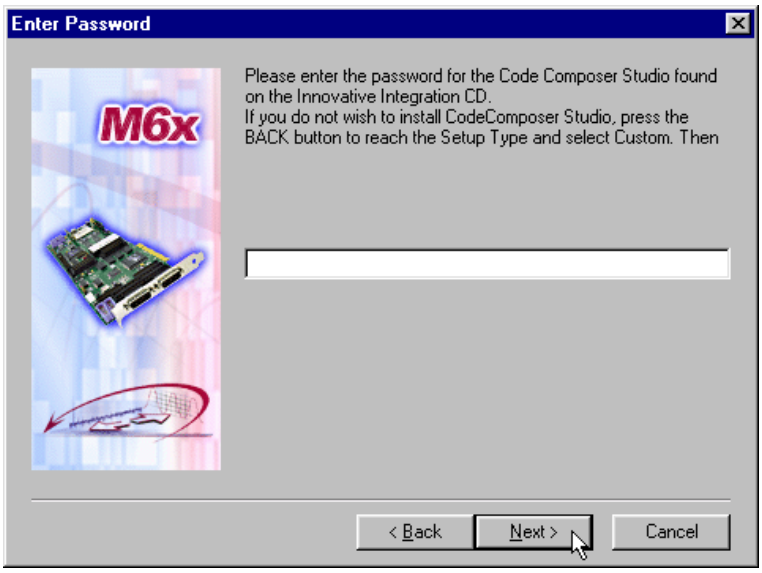
Next, the InstallShield will ask which type of JTAG you wish to install (PCI or ISA). After you have selected the type of JTAG, click **<Next>**



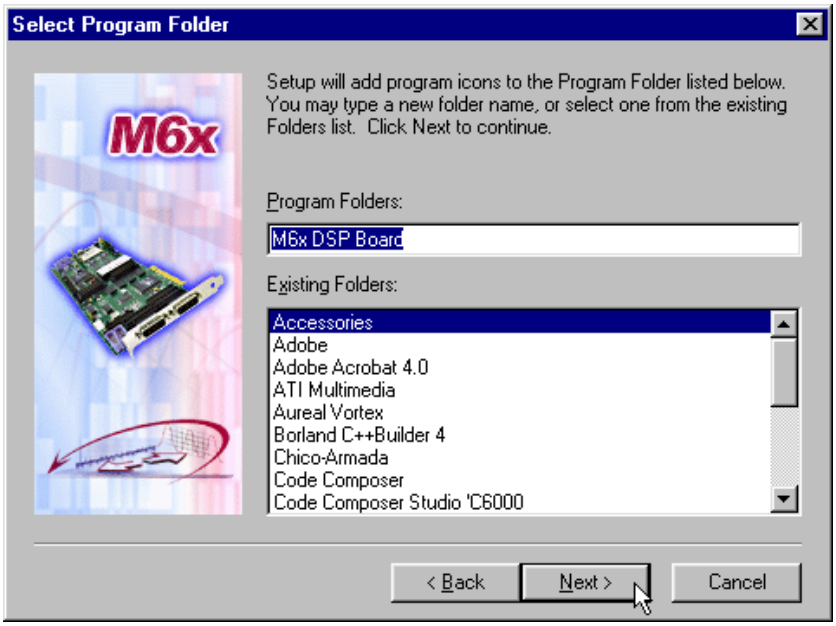
The installation will add additional components to your system in order support the development environment you are be working in. Please select the environment you are working in. If you are working in an alternate environment not listed below, select none.



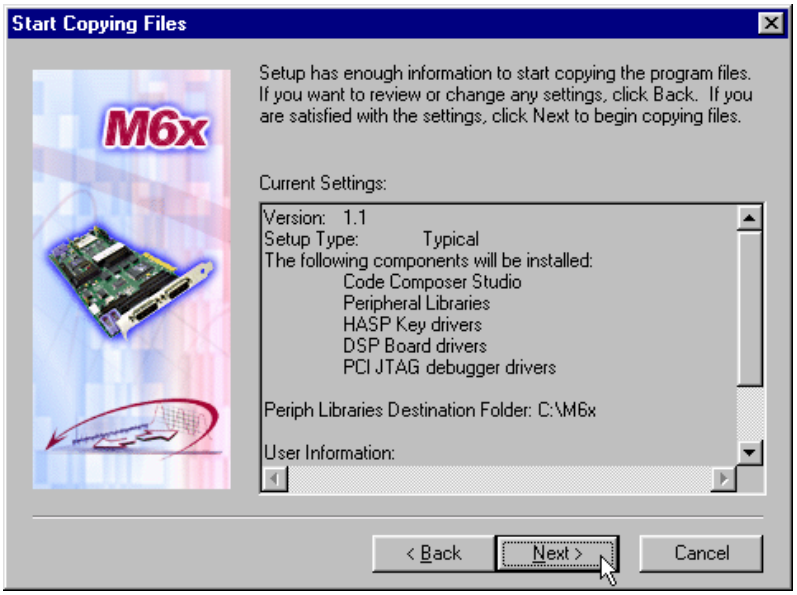
The InstallShield will now prompt you for the Code Composer Studio password. This can be found on the Innovative Integration's CD cover. If you did not purchase the Code Composer Studio with your package, go back and unselect it from the custom screen.



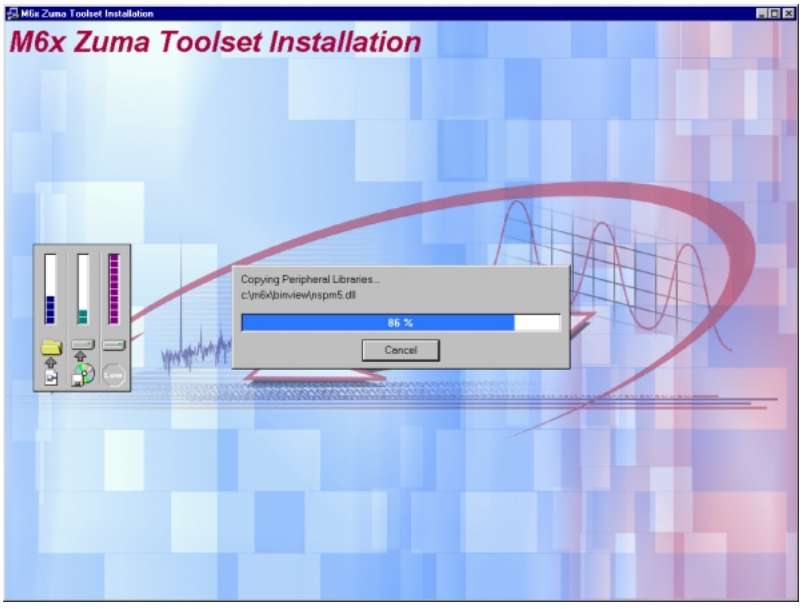
Next, you will be prompted for the program folder name to use for the board being installed. Click **<Next>** when you have made your selection.



Next you will be shown a summary of the installation selections you have made in the previous screens. If any of the items are incorrect, use the **<Back>** key to get to the screen with the selection you would like to change. When you are satisfied with the installation setup, click **<Next>** to proceed with the install.



Install Shield will now begin copying files and updating your system. A description of what is being copied should appear as shown below.



Next, InstallShield will take you through the JTAG Debugger Driver Installation.

JTAG Debugger Driver Installation

Innovative Integration Development Packages include a JTAG-based, hardware-assisted C/Assembler Source Debugger called Code Composer Studio. If you are not using Code Composer Studio or have already installed it, please skip this section.

If you have purchased an Innovative Integration hardware-assisted debugger, additional hardware, software and documentation have been included in your shipment. You should have received the following:

Item	Function
JTAG debugger board (PCI-bus compatible)	This is the PCI-bus-compatible JTAG emulator host interface board which plugs into your PC to allow communication with the target DSP over the JTAG scan path.
Target interconnect host cable	This provides an electrical connection between the host interface board and the target digital signal processor CPU.
PCI JTAG pod/pod target cable	PCI Debugger pod and pod target cable.
Code Composer Debugger package (included on installation CD)	This is the host software, which implements the debugger interface. Custom versions exist for each different DSP family - C2x, C3x, C4x and C5x.

TABLE 1. PCI Debugger Package Contents

Item	Function
JTAG debugger board (ISA-bus compatible)	This is the ISA-bus-compatible JTAG emulator host interface board which plugs into your PC to allow communication with the target DSP over the JTAG scan path.
Target interconnect cable/pod cable	This provides an electrical connection between the host interface board and the target digital signal processor CPU.
Code Composer Debugger package (included on installation CD)	This is the host software, which implements the debugger interface. Custom versions exist for each different DSP family - C2x, C3x, C4x and C5x.

TABLE 2. ISA Debugger Package Contents

PCI-JTAG Debugger Driver Installation: Windows 9X/NT

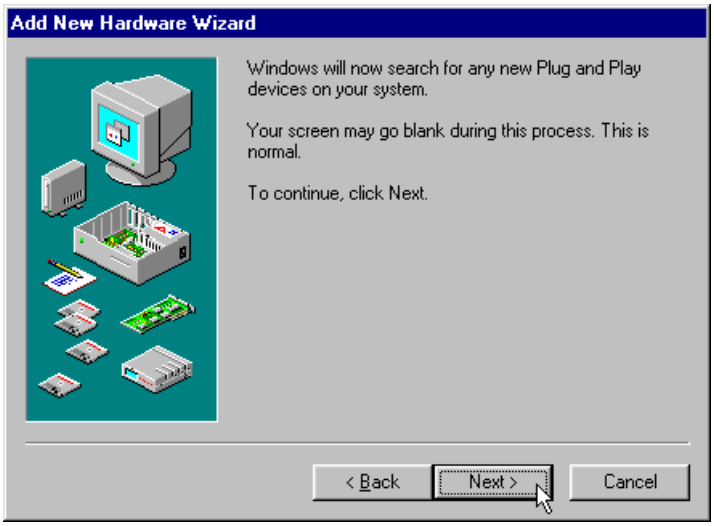
The PCI style JTAG debugger is a plug and play device. The JTAG debugger drivers are automatically installed for you and there is no acknowledgment window.

ISA-JTAG Debugger Driver Installation: Windows 9x

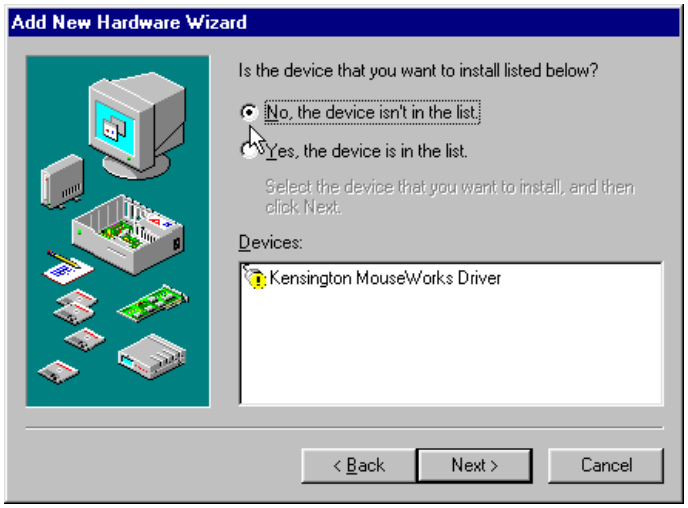
The Add New Hardware Wizard will guide you through the JTAG device driver installation. Select **<OK>** to continue.

Select **<Next>** to proceed to the next phase of the hardware driver installation.

Click **<Next>** to allow Windows to search for any new plug and play devices.

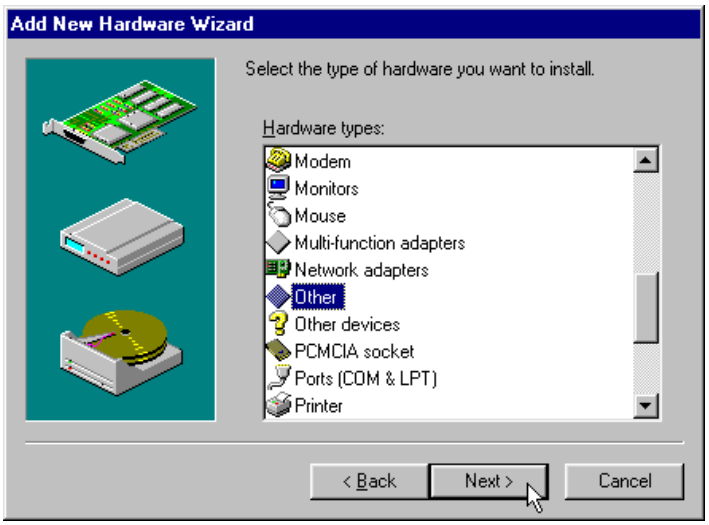


If Windows finds any plug and play devices, they will be listed in the screen below and will ask if the device you want to install is listed. Select **<No>**, and then click **<Next>** to proceed:

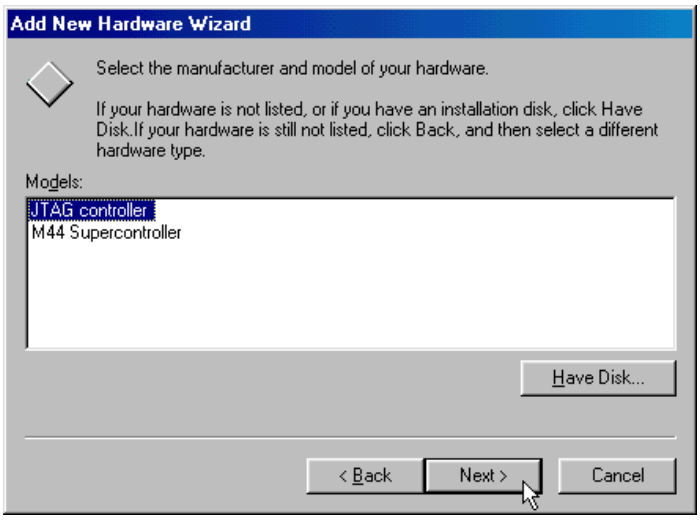


Windows will now ask if you want it to search for non-plug and play devices installed in your PC. Click <**No**>, and then <**Next**> to proceed:

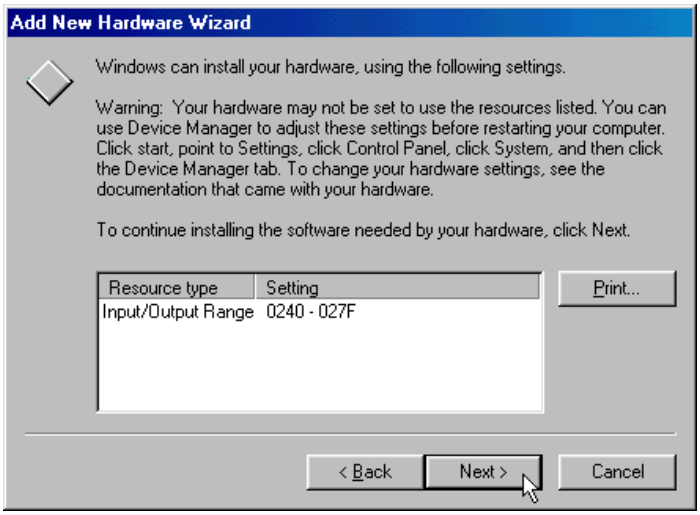
Windows will prompt you as to the type of new hardware being installed. Select <**Other**>, and then <**Next**> to proceed:



Windows will then list the installed drivers. Select the JTAG Controller device and then click **<Next>** as seen on the screen below.



The next dialog box will display the I/O address assigned to the JTAG board. It is important to make a

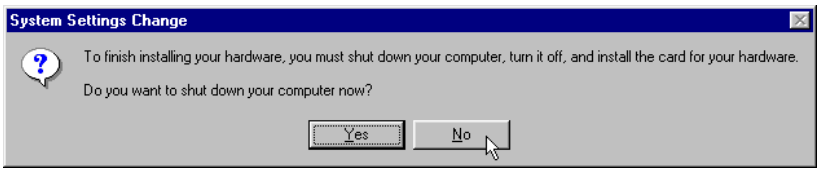


note of the **address range listed**, as it may be needed when configuring the emulator prior to JTAG hardware installation.

This information can be re-displayed at any time by simply opening the control panel, double-clicking **<System>** icon, selecting **<Device Manager>** tab, double clicking the **<Other>** category, double clicking the JTAG controller device and viewing its **<Resources>** tab.

Windows will install the drivers. When finished, Windows shows the dialog box below. Click **<Finish>** to continue

The InstallShield will now ask if you would like to shutdown your computer. Select **<NO>** to proceed with the installation.

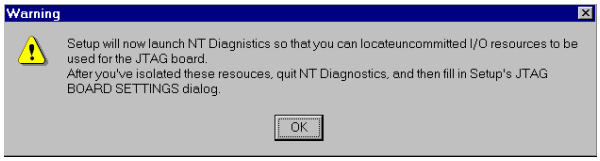


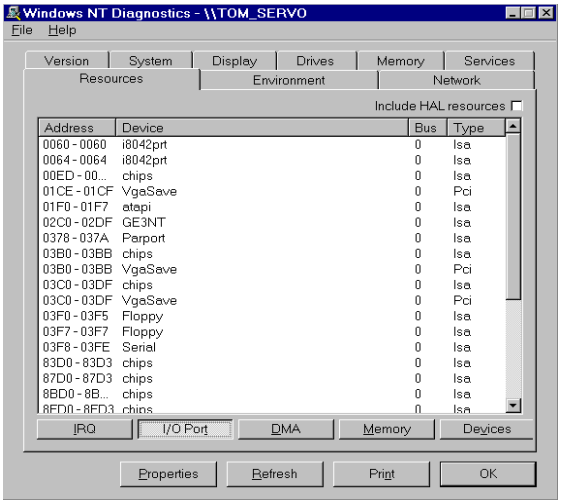
ISA-JTAG Debugger Driver Installation: Windows NT

If you are not installing on a Windows NT system, skip to the next section of this manual. The Installation will guide you through the DSP board device driver installation for Windows NT. If you are not using Windows NT, skip to the next section.

You will be prompted to launch NT Diagnostics. As with the DSP board device driver installation, select the **<Resources>** tab and then click on the **<I/O Port>** button to see all used I/O locations (see figure below). The JTAG board requires 40h bytes of I/O space. The JTAG debugger does not require IRQ or memory resources.

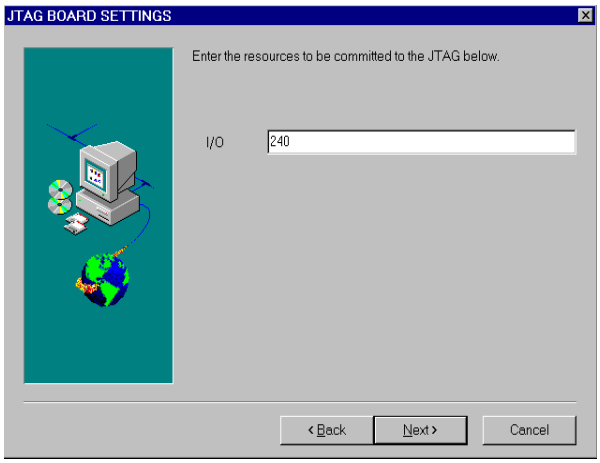
Important, Please Note: The installation of the NT Device Driver for the JTAG card requires the installing user to have Administrator rights on the system. This does not have to be the actual Administrator login, as long as the rights are the same.





Once you have located the appropriate resources, click **<OK>** in the Diagnostics dialog box.

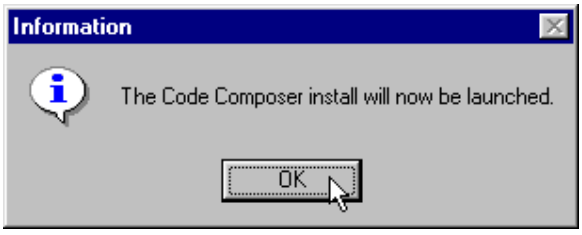
The install program will then display the default resource settings. Make sure to change the settings if they conflict with existing devices.



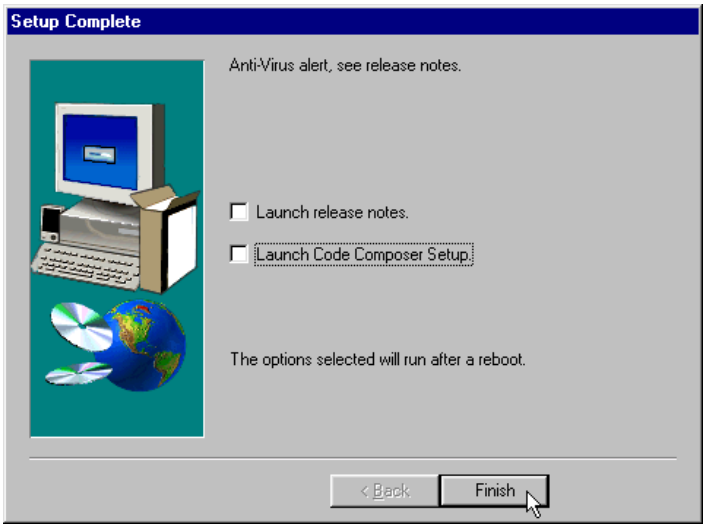
Important, Microsoft Windows NT Users Please Note: The NT Device Driver is configured to start up ***Automatically*** after it is installed.

Code Composer Studio Installation

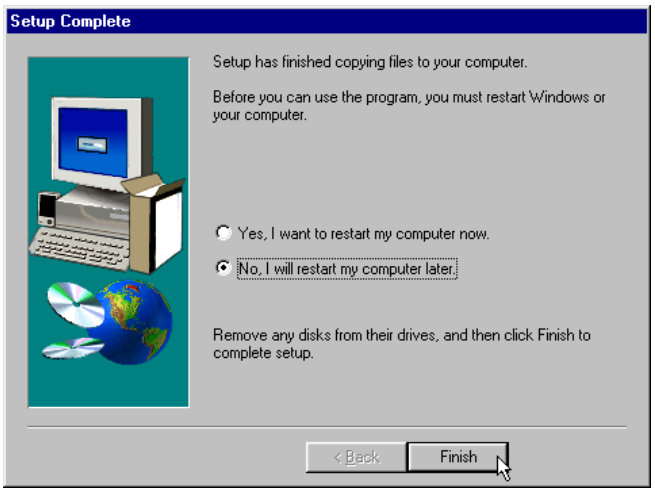
The InstallShield will now launch the Code Composer Studio installation program, click **<OK>** to proceed. Please refer to the Code Composer Studio Manual and on screen instructions for additional installation details.



The Code Composer Studio setup complete message below indicates that the Code Composer Setup program should be run to configure the device drivers. This will not be necessary since the device drivers have been automatically configured for your target hardware. The release notes can be view after the complete installation is concluded. Therefore, un-select both the "Launch release notes" and the "Launch Code Composer Setup". Click **<Finish>** to proceed with the installation.



Note: The Code Composer Studio setup complete message below suggests restarting your computer. To continue with the installation, select the "No" option and then click **<Finish>** to proceed with the M6x Zuma Tool installation



In order for Code Composer to function you must install the Code Composer Studio key onto the parallel port containing the Hasp key. The Code Composer Studio key can be plugged directly into the Hasp key.

Hasp Key Installation

The Hasp key installation utility should be automatically activated next.

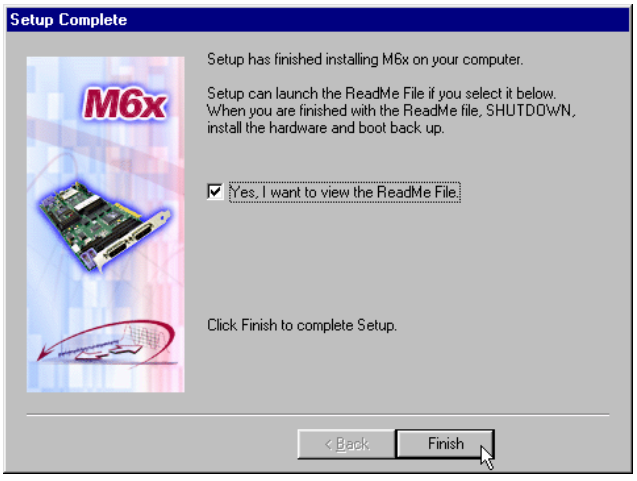
Upon completion, the Install Utility will show this screen below. It is simply an acknowledgment that the Hasp install went successfully. Your computer must be restarted before any changes made will take effect. Click the **<OK>** button, it will NOT restart your computer.

If you need at any time to run the Hasp Device Driver Installation Utility, simply run it through the InstallShield, following the installation instructions. When asked for the "Setup Type", choose "Custom" and select only the "Hasp Key Drivers" installation.

To uninstall the Hasp Device Drivers, double click the "Uninstall Hasp" option on the boards program icon. For instance, from the "Start Menu", "Start | Programs | <target board> | Uninstall Hasp". This will uninstall the Hasp Key Drivers.

End Of Installation

Now you will see the final installation screen shown below. If you would like to review the Read Me file, select the checkbox.



At this point you have finished the software installation process. Exit by clicking on the **<Finish>** button and go to the Hardware Installation section of this manual. This will conclude the Innovative Integration Development Software Package installation. Remove the CD from the drive and shutdown your computer system in preparation for installing the hardware.

Hardware Installation

The software components of the Development Package have been installed. To proceed with the Development Package Kit installation, it will be necessary to configure and install your hardware.

JTAG Emulator Hardware Installation

First, the emulator hardware must be configured and installed into your PC. The emulator hardware is described in the table below:

Type	Features
Pod-based	Uses a special ribbon cable with integrated line drivers to connect the target DSP emulation signals to the JTAG debugger card. Usable on 3.3 volt or 5 volt designs. (Including ‘C54x and ‘C6x)

PCI Pod-Based Emulator Installation

To install the PCI pod based emulator, follow the instructions below:

- Shut down Windows and power-off the host system.
- Touch the chassis of the PC to dissipate any built up static charge.
- Securely install the JTAG board into the host computer.
- Connect the host pod cable from the JTAG board external connection on the end bracket to the JTAG pod connection. Then, connect the target cable from the JTAG pod to the target DSP card connection.

ISA Pod-Based Emulator Installation

Use the following directions to install a pod-based emulator card.

If you haven't already done so, shut down Windows and power-off your PC. Set the emulator card's address to the start of the range given by the emulator device driver just installed (Input/Output Range from the JTAG Debugger Driver Installation section). The emulator address is adjusted by using a set of jumpers on the emulator board. The following diagram and table give the appropriate jumper/switch setting for the pod-based emulator board.

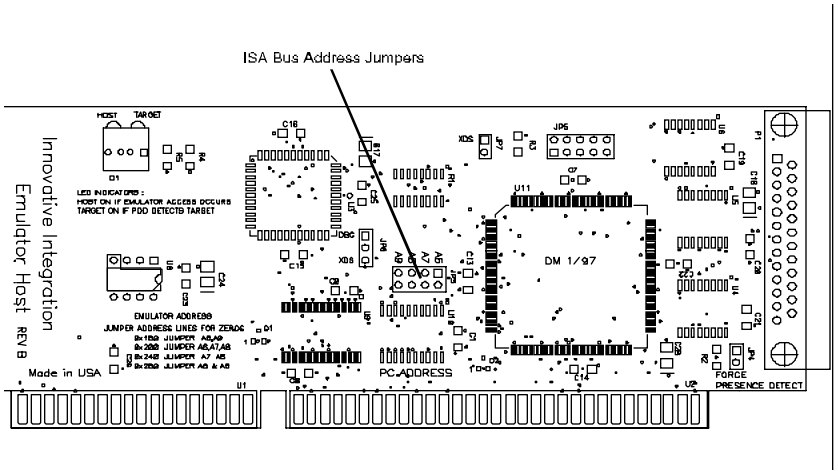


FIGURE 1. Pod Based Emulator Switch/Jumper Positions

I/O Address	A9	A8	A7	A6
0x100	ON	OFF	ON	ON
0x140	ON	OFF	ON	OFF
0x180	ON	OFF	OFF	ON
0x1C0	ON	OFF	OFF	OFF
0x200	OFF	ON	ON	ON
0x240	OFF	ON	ON	OFF
0x280	OFF	ON	OFF	ON
0x2C0	OFF	ON	OFF	OFF
0x300	OFF	OFF	ON	ON
0x340	OFF	OFF	ON	OFF
0x380	OFF	OFF	OFF	ON
0x3C0	OFF	OFF	OFF	OFF

TABLE 3. Pod-Based Emulator Card I/O Address Switch Settings

Once the address is set, install the board in the host computer and connect the pod cable to the external DB25 connector on the end bracket. Plug the pod's target connector into the target DSP card. On JTAG pods, a standard 14-pin connector has been provided.

DSP Board Installation

Innovative Integration makes DSP products that fall into three basic categories. Hardware installation directions are given below for the target card. When installing the target card:

1. Power off the host system and **touch the chassis** of the host computer system to dissipate any static charge.
2. Remove the DSP card from its protective static-safe shipping container, being careful to handle the card only by the edges.
3. Install the DSP board into an available 32-bit PCI slot in your PC.
4. Connect the JTAG debugger pod cable from the JTAG board connection to the connector (JP19) on the target board.
5. Securely install the Hasp Key (see figure below) provided with your board into a parallel port now, usually LPT1. Terminal will not run without this key.
6. In order for Code Composer Studio to function, you must install the Code Composer Studio key onto the parallel port containing the Hasp key. The Code Composer key can be plugged directly into the Hasp key.
7. After completing the hardware installation, boot up your PC.
8. The target card is plug and play, which Windows will detect it at start-up.

FIGURE 2. Hasp Key

Testing the Development Package Installation

At this point, all of the core software and hardware elements of the Innovative Integration Development Package have been installed. Through this section, <target directory> represents the target boards directory (example C:\m4x or C:\M6x). In order to test your installation, follow the instructions below.

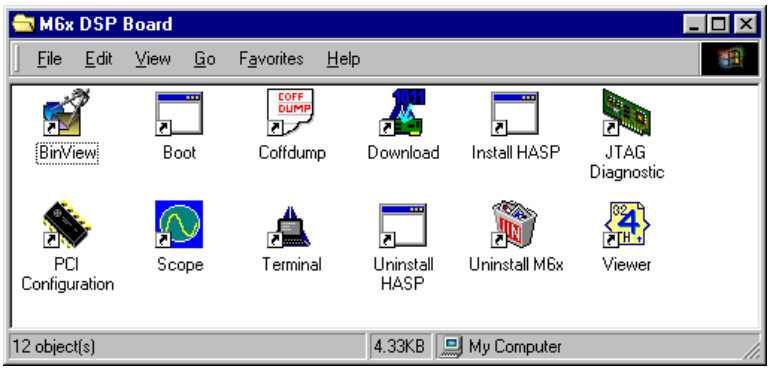
Configuring the Applets within the Development Package

Each of the Development Packages is supplied with several, standard Windows applets, which are used to perform common functions with the DSP board. These standard applets include:

Applet	Function
DOWNLOAD.EXE	Application to download a debugged DSP application to a DSP target board without using JTAG debugger.
TERMINAL.EXE	Application to act as a terminal emulator to standard I/O requests posted by the target DSP board during target executing.
COFFDUMP.EXE	Application to display memory usage of target executables.
BURN.EXE	Application to support burning application code and Talker code into FLASH ROM on FLASH-based DSP products.

TABLE 4. Host Support Applications

These applets are located in the root of the board-specific peripheral libraries and may be accessed using the Explorer or by right-clicking the **<Start>** button, clicking **<Open>**, double-clicking **<Programs>** and double-clicking (opening) the Folder associated with the DSP board. This should open a window containing icons similar to the following:



The target applets are configured in 2 ways:

Directly with the .exe file (“Directly with the .exe file” on page 36)

Through the Start menu (“Through the Start menu” on page 36)

Directly with the .exe file

Start the applet by simply double-clicking the applet from within the program group, to run it. For instance, "C:\<target directory>\Terminal.exe". This creates a .INI data file that contains the configuration information for the applet. If necessary, this .INI file can be edited (using Notepad or a text editor) to modify parameters as required.

For example, you may need to modify the Target Number of the DSP board. This is simply a handle to a DSP device used by the Windows DLL and should be zero unless you are using more than one DSP in your PC at a time.

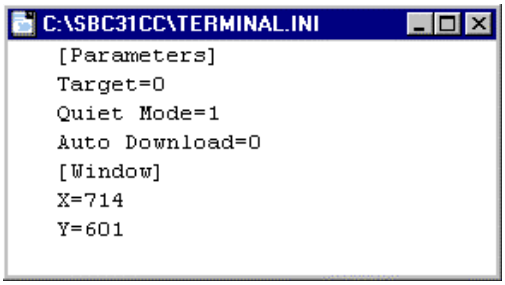


FIGURE 3. .INI File Parameters

On single-board DSP's, target zero refers to serial port COM1 and target one refers to serial port COM2, etc.

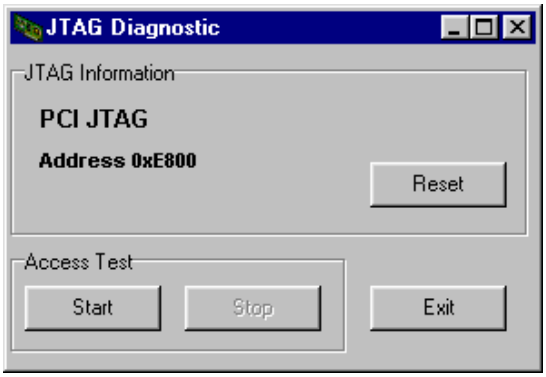
Through the Start menu

From the "Start Menu". For instance "Start | Programs | <target directory> | Terminal". If you need to modify the Target Number of the DSP Board, then you must modify the shortcut to the applet to use the correct Target Number. Bring up the Properties sheet for the shortcut of the applet by right-clicking Terminal off the Start menu.

In the Terminal properties window, select the "Shortcuts" tab. Change the Target to 1 by modifying the "Target" to read "C:\<target directory>\Terminal.exe –t 1". Click <**Apply**> and then <**Close**>.

Running the "JTAG Diagnostic" Utility

To verify that the JTAG is functioning properly, open the JTAG Diagnostic by right-clicking the <**Start**> button, clicking <**Open**>, double-clicking <**Programs**>, double-clicking on the <**target directory**>, and double-clicking on <**JTAG Diagnostic**>. The JTAG Diagnostic utility should run, seen below.



Then press the <**Reset**> button before running the test.

If the following 2 items are true:

- The JTAG card must be properly jumpered to match the Input/Output Range specified by Windows (ISA based JTAG only).

- The JTAG Diagnostic "JTAG Address" must have the same setting assigned to the JTAG device.

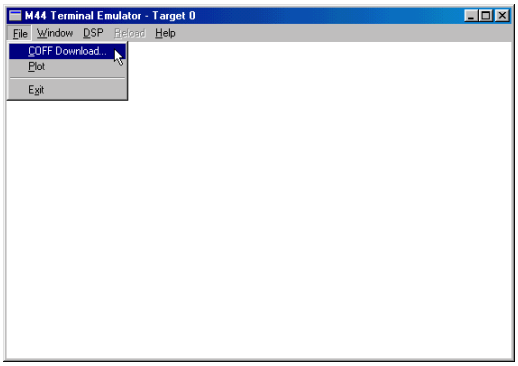
If the above conditions are all met and the JTAG is operating properly, then the START/STOP test will cause the LED on the JTAG card to blink at 0.5 Hz. Click the <**Start**> button. This should cause the LED to blink. When you are satisfied that it is operating correctly, click the <**Stop**> button.

Close the Jtag Diagnostic utility by clicking the <**Exit**> button.

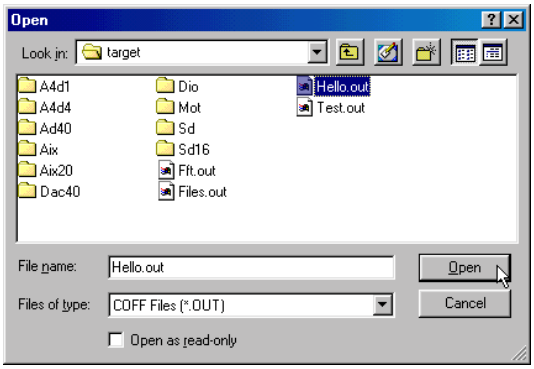
Running an Example Program using TERMINAL

Each of the Development Packages is supplied with a terminal emulator application which can be used either stand-alone or in conjunction with Code Composer Studio. The terminal emulator application is a small, Windows applet, which acts as a receptacle for standard I/O requests generated by a target DSP application. Refer to the "Support Applets" section of this Manual for detailed information on the terminal emulator.

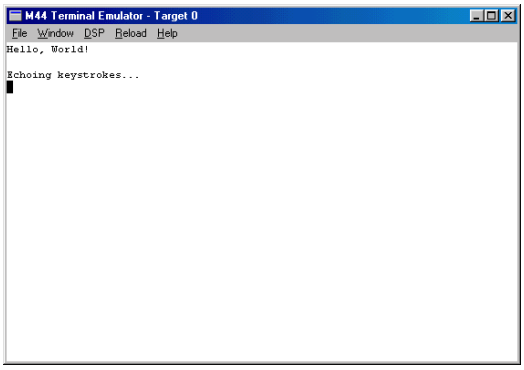
Invoke the TERMINAL utility now. If successfully started, terminal will display "Talker Init OK" in its client window. If the Talker fails to start, refer to the Troubleshooting section of this manual. Iterate this step until Talker initializes successfully. You should see a window similar to the following:



Select **<COFF Download>** from the **<File>** menu, to begin downloading a program to the target DSP. This will open a dialog box from which you can select a target DSP program to run. The examples can be found in the C:\<*board directory*>\examples\target\ directory:



Select **HELLO.OUT** from the file list and click **<Open>** to download and run the classic "Hello World!" program to the target DSP. The terminal emulator should display "Hello World!", as shown below:



If so, close the "Terminal" program and proceed to running the "scope" test. Otherwise, refer to the Troubleshooting section of this manual for the most frequently asked questions and solutions.

Running the "Scope"

Note: This test is for all boards *except* the SBC32.

Bring up the Scope through the “Start Menu”. For instance, Start | Programs | <target board> | Scope.

For PCI-bus-based products:

In the ".OUT File to Download" box, press the <**...**> button which will open a dialog box from which you can select the scope.out file. This is found in the c:\<*board directory*>\examples\host\scope\dsp\ directory.

Select scope.out from the file list and click <**Open**>. Make sure you have the correct "Target" value selected (usually 0).

Click <**Open**> on the scope dialog box - this will enable the <**Start**> button.

Press the <**Start**> button to start the scope. A Sine wave should be displayed.

Press <**Stop**> when you are done.

For ISA-bus-based products:

The scope should automatically display a sine wave. There may be a delay initially.

To exit the scope, click <**Exit**> on the scope dialog box.

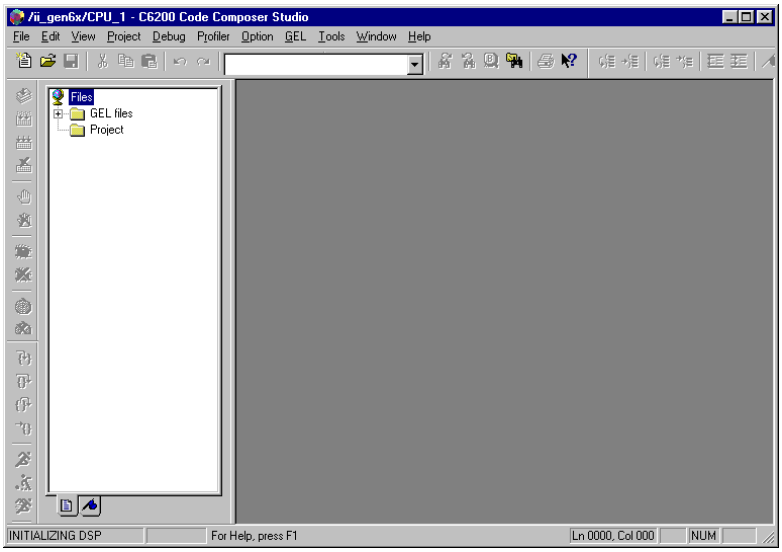
Proceed to Code Composer Testing, below.

Testing the Code Composer Debugger

If you will be running an application which employs standard I/O (i.e. most of the II example programs), start Terminal.exe. (Refer to **Running an Example Program using TERMINAL** for instructions) In Terminal, click Window | Always on Top. This will force the Terminal to always show. Note that Terminal must be launched prior to Code Composer Studio because Terminal physically resets the target DSP board during its initialization, which disrupts the JTAG hardware used by Code Composer Studio.

Next, open the Code Composer folder by right-clicking the <**Start**> button, clicking <**Open**>, double clicking <**Programs**>, and double clicking (opening) the Code Composer Studio folder. This should open a window containing icons similar to the following:

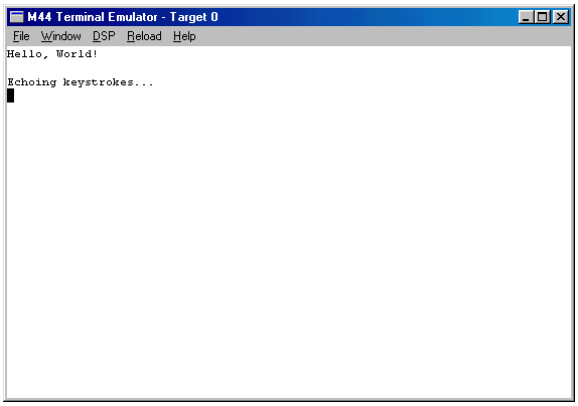
Double-click on the Code Composer icon “CC Studio” to launch the debugger. You should see a window similar to the following:



If you do not see the above window, refer to the Troubleshooting section of this manual for the most frequently asked questions and solutions.

If you do see the above window, load and run the HELLO.OUT target application as described below from Zuma Toolset Examples\Target directory. To load, use File | Load Program and to run, use Debug | Run Free.

The DSP target application should run, displaying “Hello World!” in the terminal window:



You have successfully run your first DSP program from within the Code Composer Studio Source Debugger environment! Refer to the Code Composer Studio documentation for complete instructions on how to take advantage of all the features within the debugger software.

Troubleshooting Installation Problems

Most Commonly Asked Questions

This section includes answers to some of the most commonly asked question relevant to installation and initial testing. If after troubleshooting, components of the Developer’s Package still do not operate correctly, contact Innovative Integration for technical support.

I already had a licensed copy of the TI tools, so I omitted them from the Development Package. Whenever I attempt to compile, assemble or link a program from within Code Composer Studio, the build window shows “Bad command or filename” errors.

Edit your **AUTOEXEC.BAT** file to add the directory containing your TI toolset to your default path, i.e.:

path = c:\windows;c:\windows\system;c:\fltc

Code Composer Studio won't start. It shows a dialog box that says, "Valid Hardware Key could not be detected. Please insure the hardware key is fastened securely to your parallel port."

Make sure the Code Composer Studio hardware key is attached properly to the Hasp Key. Use the thumbscrews to securely attach the key to the parallel port. Also, make sure you have the correct hardware key for your board.

Code Composer Studio won't start. It shows a dialog box that says, "Can't initialize target DSP. Trouble with JTAG controller. Please insure the I/O port is set properly.

There are several common reasons for this error. Verify each of the following:

- You have properly configured the ISA JTAG debugger board according to the I/O assignment produced by Windows during the ISA JTAG device driver installation and that all jumpers are properly oriented for communication with your target.

- Verify that your JTAG cables are properly connected to the host and DSP target board.

- ISA DSP board users: Verify that you have properly configured the DSP target board according to the I/O and interrupt assignments produced by Windows during the DSP board device driver installation.

- Stand-alone DSP board users: Verify that the DSP board is powered up and the supply voltages are correct. If you are using a serial mouse, change the target number setting in the terminal.ini file (found in the board's root directory).

- You may have selected the incorrect driver for your DSP target within the Code Composer Studio setup utility. If you are using a multiprocessor target, check the multiprocessor settings as well.

- When using the pod-based debugger with an JTAG pod, make sure the target is set up to provide the 'C3x H3 clock (see DSP card *Hardware Section* for details).

- Verify that the JTAGDIAG "Access Test" passes. Launch the "JTAG Diagnostic" from the program folder of the board and click the <**Reset**> then <**Start**> button

- Check for the proper and most current DLL driver available.

- Verify that DLL in being used is as new as the Innovative Intergration's website (www.innovative-dsp.com) version.

When I attempt to start Code Composer Studio, my PC "hangs" and won't respond to the mouse or keyboard.

If you are using a C3x-based DSP board, insure that the JTAG cable is properly connected between the debugger board's C3x JTAG connector and the target DSP board's JTAG connector.

For all other targets, insure that the JTAG board clock select is configured for OSC and that the on-board oscillator is seated in its socket.

I have checked and re-checked the settings for my JTAG board and it's connections to the DSP target, but Code Composer Studio still won't start.

The ISA/PCI card edge connector on the JTAG board may be dirty. Clean the ISA/PCI card edge connectors for the JTAG board using a pencil eraser until the edge connector is free of any film or residue.

I can't seem to load any of the Code Composer Studio workspaces for the Development Package example programs.

The project workspaces (**`*.WSP`** files) were created for proper execution when the DSP board directory exists on the **`C:\`** drive. If you installed your DSP board directory onto another drive, you will have to recreate each of the project workspaces. However, it is possible to edit each of the project make files (**`*.MAK`**), modifying all drive letter designators in order to allow them to work on another drive.

Code Composer Studio appears to operate properly (I can load, execute and step through programs), but standard I/O doesn't appear on my terminal window when I run the example programs.

Insure that the `TERMINAL` applet is configured to communicate with your target board. For single-board users, the target number corresponds to the PC com port being used (target 0 = COM1, target 1 = COM2, etc.). For all other targets, the target number is usually zero. You may need to edit the applets `.INI` file (using Notepad) in order to manually adjust the target number. The `.INI` file is located in the `II_BOARD` directory.

I have installed a PCI-based DSP card and now my PC won't boot.

You do not have an available, uncommitted IRQ for use by the PCI card. Enter the system BIOS setup and reserve an IRQ for use by the DSP board.

I have installed a PCI-based DSP card and my PC boots. But an ISA adapter card in my system (network card, etc.), which used to work fine, is no longer operating.

The PCI BIOS has assigned the DSP board an IRQ that was already in use by the ISA board. Enter the system BIOS and reserve an IRQ for use by the DSP board.

Our host application is not Windows-based. We're using DOS, UNIX, OS2, etc. for our host environment. How can we develop and debug my target application?

Install Code Composer Studio and the TI tools onto a Windows-based PC and umbilical over to the DSP board installed in a second machine that is running under the "foreign" operating system. You will not be able to run TERMINAL and DOWNLOAD under the foreign OS, but the Windows-based system can be used to develop and deliver code to the target DSP over the JTAG link.

I get a "Talker didn't start!" message when attempting to download to my single-board DSP from within TERMINAL or DOWNLOAD.

- Insure that the applet is configured to communicate with your target board. For single-board users, the target number corresponds to the PC com port being used (target 0 = COM1, target 1 = COM2, etc.). For all other targets, the target number is usually zero. You may need to edit the applets `.INI` file (using Notepad) in order to manually adjust the target number. The `.INI` file is located in the `II_BOARD` directory.

- For Stand Alone boards, verify that the board is getting power and the serial cable is well seated. Also, make sure all external reset sources connected to the board are not stopping the card from running.

- Verify that the jumpers on the card are set to the factory defaults.

- Check that the COM port in use is enabled at the BIOS level.

- Check that the port is enabled and available from within the Windows Device Manager

We are planning to use Visual Basic for our host application and want to know if we can access your host DLL functions?

Yes, the DLL functions are accessible with this tool.

After installing a stand-alone board, my serial mouse no longer works.

Terminal is using the same target number as your mouse (target 0 = COM1, target 1 = COM2, etc.). Change the target number setting in the `TERMINAL.INI` file in the board's directory.

After installing a stand-alone board, my computer won't reboot, I get a keyboard error at boot up, or Windows hangs at start up.

Your DSP may be stuck in a bad state. Turn off your computer. Remove power to the board. Turn on your computer and wait for windows to boot (if prompted to start in safe mode, ignore the message and do a normal boot). Once windows has started, power up the board.

I have downloaded a new driver for my board from you FTP site. How should I install it?

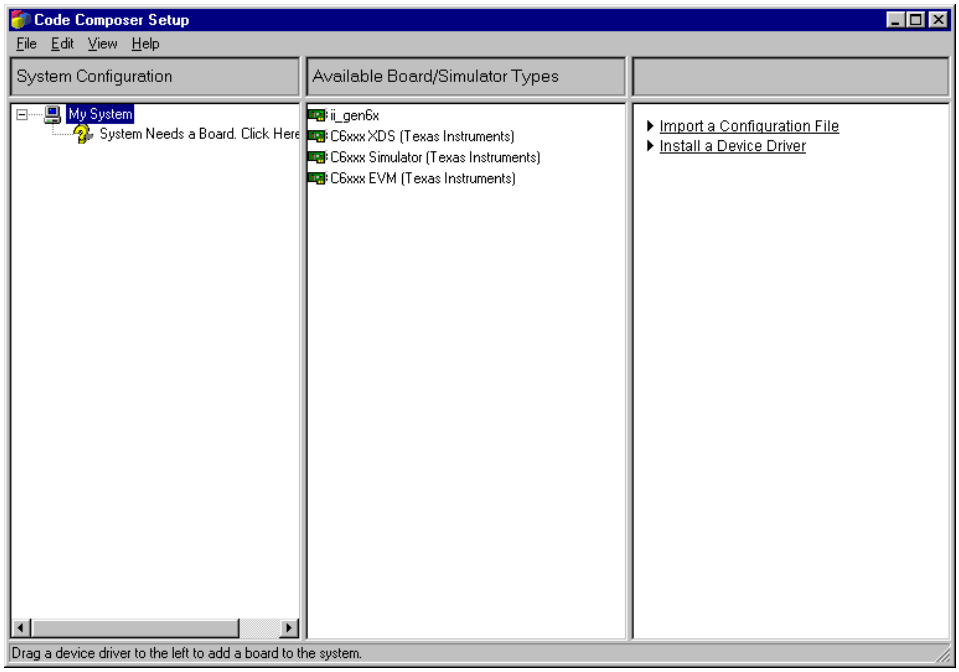
Re-run the driver installation as documented earlier in this document. Your old driver will be overwritten during the installation.

Code Composer Studio Troubleshooting

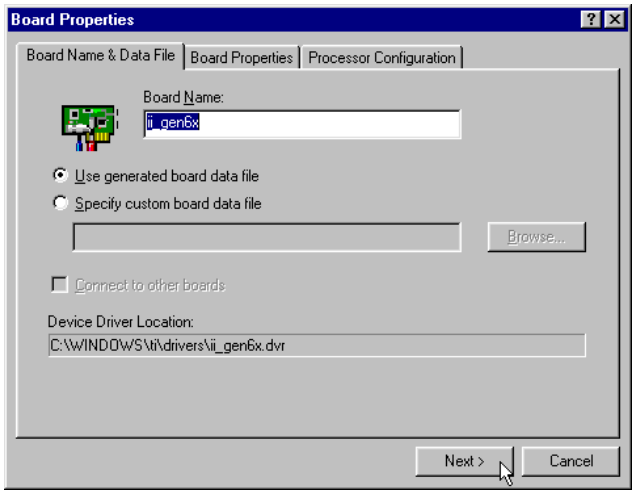
If Code Composer Studio did not install or operate properly, it may be helpful to refer to the Code Composer Studio instruction manual for additional information.

Code Composer Studio requires third party device drivers to be installed along with the executable application in order to support Innovative Integration's debugger hardware. Therefore, if these device drivers did not install properly, Code Composer Studio may need to configure for use with the Innovative Integration DSP board you have purchased before being run.

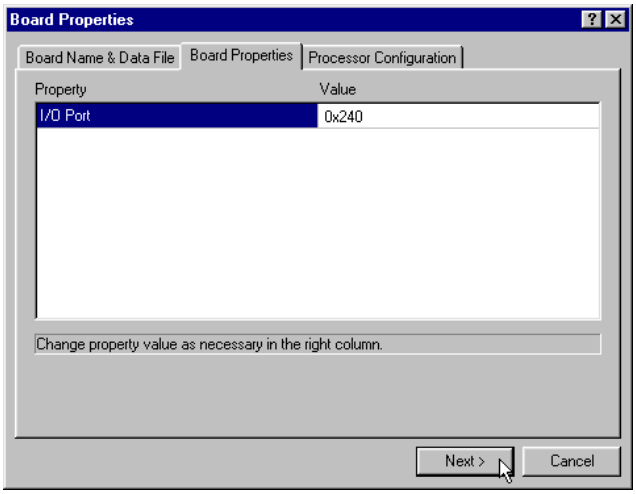
The Installshield should have already installed the Code Composer Studio driver for the DSP board. Although if it had not, then open the Code Composer Studio folder by right-clicking the **<Start>** button, clicking **<Open>**, double clicking **<Programs>,** and double clicking (opening) the Code Composer Studio folder. Then double click on the Setup CCStudio to launch the Code Composer Studio Setup program seen below.



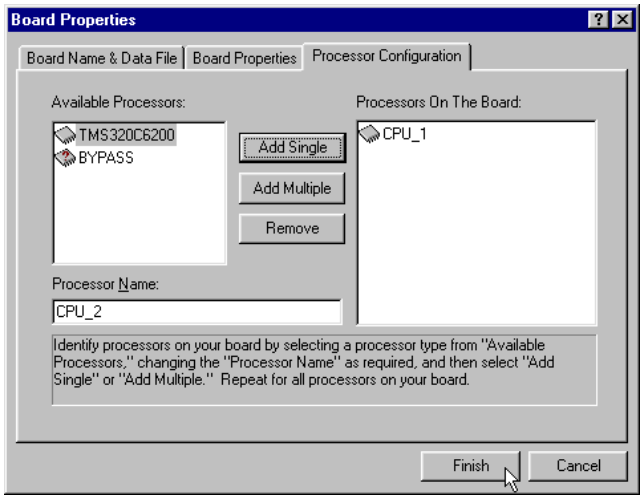
Next, right click and drag the II_Gen6x icon from the “Available Board/Simulator Types” column to the “System Configuration” column. This will open the following window.



The board name can be change in this window, but it is recommended that you keep the default name for clarity. Once you have set the board name, click **<Next>** to proceed.

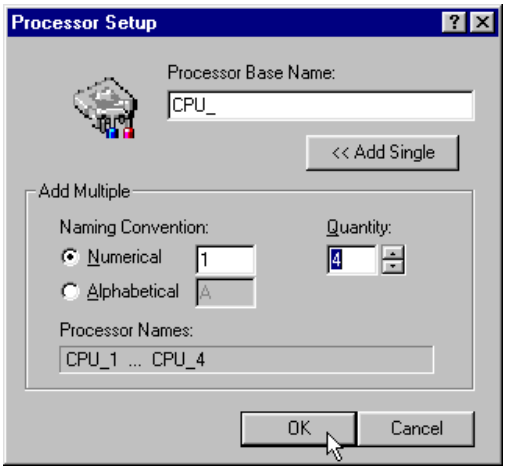


Verify and/or change the board properties I/O port address value in this screen to the address assigned to the JTAG by Windows during the JTAG debugger driver software installation phase and then click **<Next>**. (The Windows assigned JTAG I/O port address can be displayed by the device manager)

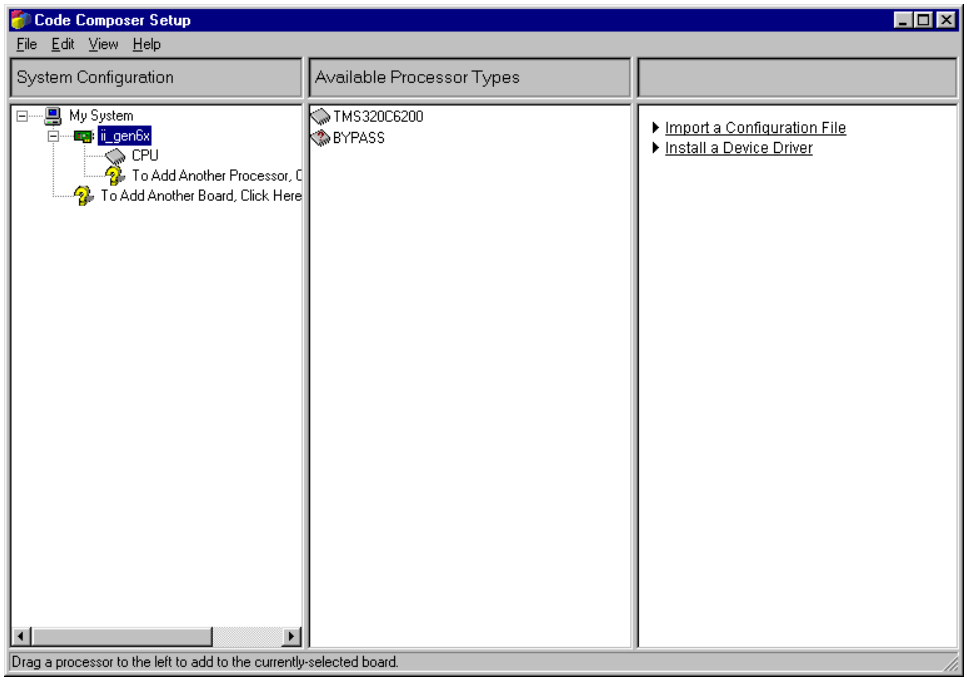


Add a single processor to the board by clicking the **<Add Single>** button and then click **<Finish>** to conclude.

If the target is a multiprocessor DSP card, click the **<Add Multiple>** and enter processor base name for the target hardware as seen below. Refer to the Code Composer Studio manual for more information on the use of multiprocessor debugging.



Please note that the number of processors entered in the scan path list **must** be equal to the actual number of processors in the emulator scan path. Note also that order of the CPU IDs must match the order of the CPU's in the JTAG scan path. This is accomplished by entering the identifiers in what appears to be reverse order (with cpu_2 before cpu_1 and cpu_3 before cpu_2) in the *Processor List*:



The Code Composer Studio Setup screen should look similar to the one above. Save the setup setting and exit the Code Composer Studio setup screen. Code Composer Studio debugger installation is now completed.

Verify Environment Variables

The II Zuma package makes use of DOS environment variables in order to locate header files, code generation executables, etc. The installation process sets these variables to the settings shown in the ReadMe file at the time of installation. Be sure to verify the Environment variable settings, especially if you have installed the TI Compiler/Assembler after the board, before trying to use the board. *Note:* upgrading from previous versions or when mixing development components from II and other vendors, problems can arise.

To verify the settings, go to a DOS Prompt and type "SET". This will write to the screen the variables and their values. Check these values against the ReadMe file. The ReadMe file can be found in the root of the board directory – i.e. C:\<board directory>\ReadMe.txt

To modify the variables under Window 9x, reset these variables in your **AUTOEXEC.BAT** file. Under WinNT, set these variables by opening Control Panel | System Icon | Environment tab. Note: place

all entries under System Variables: not under User variables section. Double click the variable name to modify: edit box (i.e. c_dir), then enter value of environment variable in Value: edit box (i.e. c:\fltc). After finishing an entry press 'Set' button to add it to current settings. Note, unlike Win95/Win98, after making changes to environment variable settings they take effect immediately, you do not have to reboot.

Below is a description of the Environment variables that are modified by the installation of an Innovative Integration Board installation:

DOS Environment Variable Name	**Products Affected**	**Description**
II_BOARD	CodeWright/Peripheral Libraries Directory	Board specific library directory - Used for infrastructure
C6x_C_DIR	All TI C Compilers All II peripheral libraries	Code generation tools directory - Search path for the compiler
C6x_A_DIR (optional)	All TI Assemblers	Code generation tools directory - Search path for the assembler
D_SRC	All Debugger products	Optional JTAG Debug Directory - Search path for the debugger
D_DIR	All TI Debuggers	Optional JTAG Debug Directory - Path to the executable for the debugger
PATH	All II products All Compilers/Assemblers	Dos search path (new entries are added to the path)

Multiple Board Support

Multiple target boards of the same type may be installed in the same system with full development software support (the only exception being the JTAG debugging support under Code Composer Studio for multiple 'C3x targets. Since the modified JTAG standard used on the 'C3x processors does not support multiple processor debugging, Code Composer Studio may be used with only one 'C3x target at a time). Multiple copies of the support applications may be run simultaneously, each communicating with different targets, to provide parallel support for multiple target boards. Follow the instructions below to set up support for more than one target:

1. Go through the normal installation of the support software per the instructions above.

2. For each target board, make a Windows shortcut icon for each application, which must be used simultaneously. For example, if the system has three target boards installed and the user wishes to use the COFF downloader and terminal emulators independently with each board. Then make three shortcuts each for the two applications and label them "COFF Downloader Target 0", "COFF Downloader Target 1", etc. To make a shortcut icon, open the "My Computer" desktop icon and open the drive and installation directory where the development tools were installed. Right click on the application for which the shortcut will be made, and select "Create Shortcut". A new icon will appear in the folder window, labeled "Shortcut to [APPLICATION NAME]". Rename the icon appropriately by right clicking and selecting the "Rename" menu entry, then entering a new board-specific name, such as "COFF Downloader for Board#1". Optionally, the shortcut may be dragged onto the desktop and the file folder closed to clear display space.

3. Once the shortcut copies have been made for all instances of the application(s) for each target, the shortcuts must be customized to point to their respective target boards. This is accomplished by adding command line switches to the Properties dialog box for each shortcut. Right click on each shortcut and select the "Properties" entry to open the Properties dialog box. Select the "Shortcut" tab and edit the "Target" text box. Add the target number override switch (-t) followed by a space and the target number of the board with which this instance of the program will communicate. To find out each board's target number, use the `FIND` utility (described below). For example, if the system has two targets installed, one at target number 0 and one at target number 1, the shortcut for the first board's COFF downloader would have a "Target" entry of:

   [install directory]\DOWNLOAD.EXE –t 0

   and the second board's COFF downloader shortcut would have an entry of:

   [install directory]\DOWNLOAD.EXE –t 1

   Additional switches may be specified in the "Target" text box to further modify the application's individual behaviors. See the support application's descriptions below for complete details on the switches available for each application.

Note: The command line switches, specified in the shortcut properties box, act as overrides to the default behavior selected in the configuration utility. Any switches NOT specified in the shortcut properties dialog box will cause the applications to revert to the global configuration selected in the configuration program. For example, if the user selects the Automatic Download feature in the configuration utility and specifies a filename, then all shortcuts created for the COFF downloader will automatically download that file on start-up. If one of the shortcuts specifies a -d[FILENAME] switch in its property box, then that shortcut will download the specified filename on start-up, rather than the default application selected in the configuration utility.

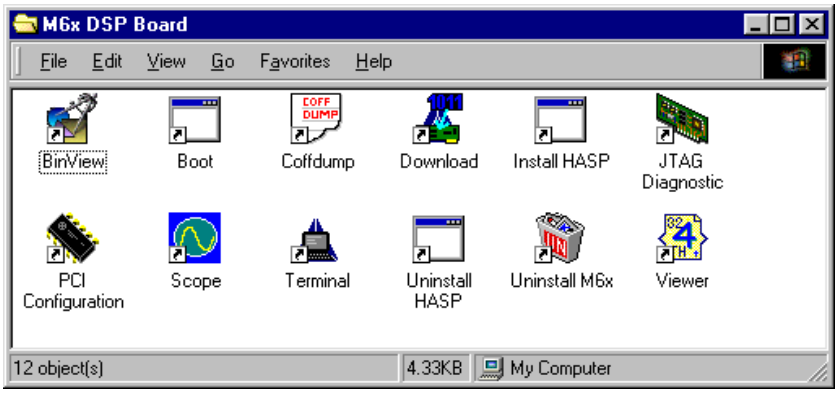
Uninstall Process

The uninstallation process is quite simple, and it is different for Win95/Win98 than for Windows NT.

Windows 95/Windows 98 Uninstallation

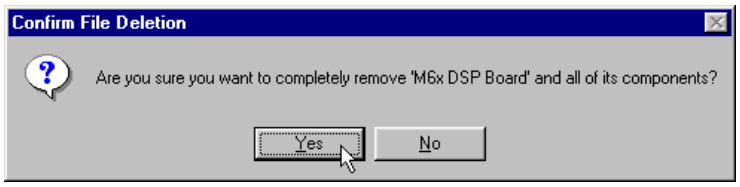
The uninstallation process consists of using the target board uninstall utility to remove the software and editing the autoexec.bat file to remove environment variables.

To uninstall the software, right-click the **<Start>** button, clicking **<Open>**, double-clicking **<Programs>**, and double-clicking the Folder associated with the DSP board. This should open a window containing icons similar to the following:

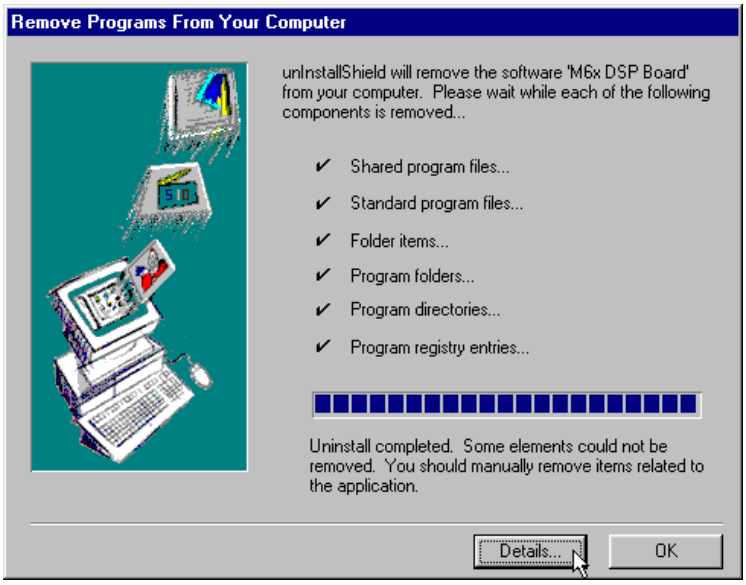


You will first double-click the Uninstall HASP icon and the following screen will appear. Click **<OK>**.

You will then double-click the **uninstall <target board>** icon and you will be asked if you really want to remove the <target board> DSP board and all of its components. Click **<Yes>**.



The program and its components will be removed and the screen below will be shown.

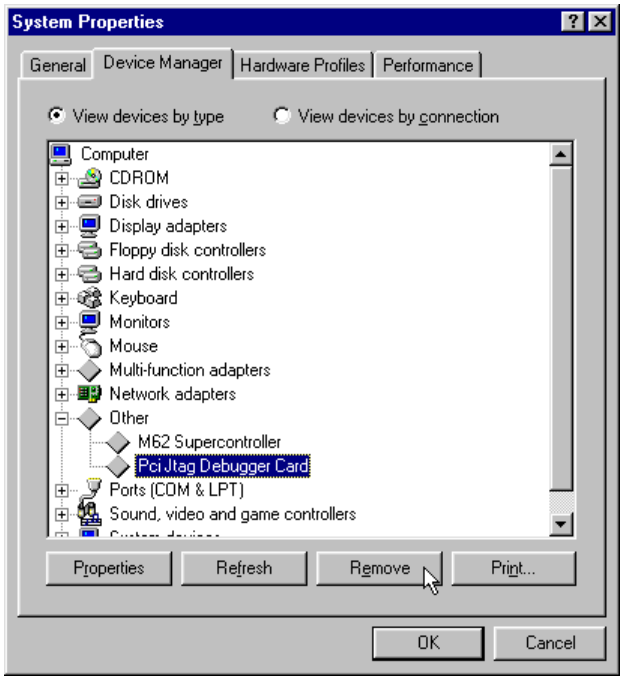


When it is done, it will let you know if any problems occurred. If there was a problem, view the **"Details"** to see what was not removed. You should then manually remove any files that were not automatically deleted.

Next, you should remove environment variables that were added to the autoexec.bat. Remove the following variables: "<target board>_a_dir", "d_src", and "ii_board". Now remove any board related paths from the "<target board>_c_dir" and "path" variables.

The JTAG device must be remove from the Device Manager as follows:

From the **Control Panel**, double-clicking on the **System icon**. Click on the **Device Manager** tab. Find the **Other** directory, and click the board you wish to uninstall (JTAG Controller). Then click the **<Remove>** button as shown below.



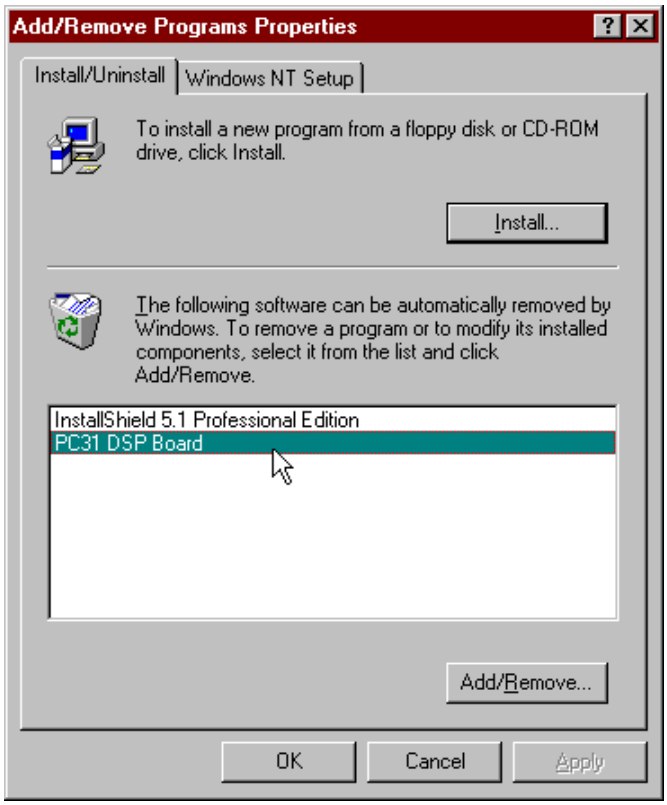
Confirm the device removal by clicking the **<OK>** button and then restart your computer system for the changes to take affect.

Note: The Code Composer Studio has not been uninstalled. To uninstall this software refer to the uninstallation instructions in the Code Composer Studio manual.

Windows NT Uninstallation

The uninstallation process consists of using the "Add/Remove" windows utility to remove the software and using the regedit utility to remove environment variables.

First, open the **"Add/Remove Programs"** utility in the **Control Panel**. Then highlight the board and click <**Add/Remove**>.



You will then be asked if you really want to remove this program. Click <**Yes**>. Next, the program will be removed. When it is done, it will let you know if any problems occurred. If there was a problem, view the "Details" to see what was not removed. You should manually remove any files that were not removed.

Next, you should remove portions of the environment variables that were added. **Remove any board related paths** from the "<target board>_c_dir" and "path" variables. For instance, if the <target board>_c_dir currently has "c:\<target board>;c:\<target board>\include\target;c:\fltc" as it's value, remove the "c:\<target board>;c:\<target board>\include\target;" portion. To accomplish this, open Control Panel | System Icon | Environment tab.

CHAPTER 3

Integrated Development Environment

The C Developer's Package consists of several software tools, integrated to work together to provide a complete DSP design environment for Innovative Integration DSP boards. This section discusses the tools included in the development package and gives descriptions of each applets features and use. A brief introduction is given regarding the software programs provided and their use within the Developer's Package. The user is referred to the individual manuals accompanying these software products for complete documentation.

The Texas Instruments C Compiler Toolset

The C compiler supplied with the Developer's Package is the Texas Instruments (T.I.) Floating Point C Compiler toolset for the DSP target board. The compiler runs under Windows as a cross compiler, generating executable applications for the DSP processor which are then downloaded and executed using the other tools in the Developer's Package. The compiler is ANSI C compatible and supports nearly all standard C functions. Additional libraries provided with the Developer's System include C standard I/O and peripheral drivers for the A/D, D/A, bit-I/O and timers. Assembly language may also be mixed with C code for higher performance where required.

Typical application programs will consist of one or more C (`.C`), header (`.H`), and Assembly language (`.ASM`) source files, as needed. Additionally, target program generation requires use of a linker command file (`.CMD`) which specifies the memory map for the target and optionally includes commands defining the libraries to be linked into the final application.

Users of the Code Composer Studio editor/debugger will also employ make (`.MAK`), workspace (`.WSP`) and special Code Composer-specific script files (`.GEL`). The example programs included in the Developer's Package illustrate the use of these files also and give example files to use as a basis for custom DSP applications.

C Compiler Toolset Usage

The C compiler may be run directly from a DOS Prompt window under Windows 95/98/NT as described in the TI toolset documentation. Also included in the installation directory are batch files useful for manually rebuilding applications programs within the DOS environment. `COMPILE.BAT` and `ASSEMBLE.BAT` are batch files which will re-compile/reassemble a C or Assembly source file (respectively) specified as a target parameter to these batch files. The `LINK.BAT` will invoke the TI Linker to link several object modules to create a target executable (`.OUT`) file, consuming a linker command file (`.CMD`) as a parameter.

Code Composer Studio

Code Composer Studio is a flexible, high-performance, integrated code generation environment developed by Texas Instruments and bundled into the Innovative Integration Zuma toolset. For complete documentation to the features of Code Composer Studio package, refer to the accompanying *Code Composer Studio Manual* provided. If the user wishes to compile outside of Code Composer Studio (or has not purchased the package), these make files may be used from the DOS command line to rebuild individual project files or the entire target file set.

Editor

Code Composer Studio supports code editing and emulates the most popular editing packages (CUA, etc.). Code Composer Studio is a Window editor. Custom DSP code development can take place entirely within the Code Composer Studio environment using its project management tools to place source files, libraries and linker command files into projects (.MAK) in order to build executables. The example programs included in the Developer's Package each have a Code Composer project file (`.MAK`) associated with them which may be used to re-compile the example.

Debugger

Code Composer Studio is a software program for high-level TI C and Assembly Language debugging which supports high-performance, JTAG or MPSD-based hardware assisted debugging directly on the target DSP to gain access to the internal register set, peripherals, and bus of the target board in order to load, run, and debug applications. Also integrated into the Code Composer Studio software package is a code management subsystem for editing files as well as creating and compiling DSP projects.

CHAPTER 4

Support Applets

The Developer's Package includes four support applications supporting general DSP development: the terminal emulator (`TERMINAL.EXE`), the COFF file downloader (`DOWNLOAD.EXE`), the COFF file display utility (`COFFDUMP.EXE`) and the FLASH prom programming facility (`BURN.EXE`). This section describes the functionality of each of the applications and their use within the development system.

The functions provided by each of the applications may be configured through menu selections available within each of the applets themselves. Generally, parameters governing the behavior of each applet are stored in program-specific `.INI` files, located in the directory from which the applet is invoked. See the discussion below for applet-specific parameters.

The Terminal Emulator

The terminal emulator provides a C language-compatible, standard I/O terminal emulation facility for interacting with the `stdio` library running on the DSP processor. Display I/O calls such as `printf()`, `scanf()`, and `getchar()` are routed between the DSP target and the Host terminal emulator applet where ASCII output data is presented to the user via a terminal emulation window and host keyboard input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems, and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation.

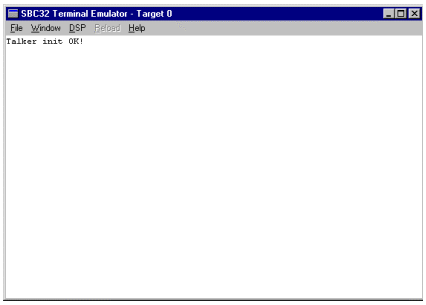


FIGURE 4. Terminal Emulator Applet

The terminal emulator is straightforward to use. The emulator will respond to stdio calls automatically from the target DSP card and should be running before the DSP application is executed in order for the program run to proceed normally. DSP program execution will be halted automatically at the first stdio library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking, except on stand-alone SBC targets. stdio output is automatically printed to the current cursor location (with wraparound and scrolling), and console keyboard input will also be displayed as it is echoed back from the target.

The terminal emulator also supports Windows file I/O using the library routines `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fseek()` and `fflush()`. Refer to the Appendix for prototypes and usage of these library functions as their usage is not 100% ANSI compliant.

Terminal Emulator Menu Commands. The terminal emulator provides several menus of commands for customizing its functionality. The following is a description of each menu entry available in the terminal emulator, and its effects

.

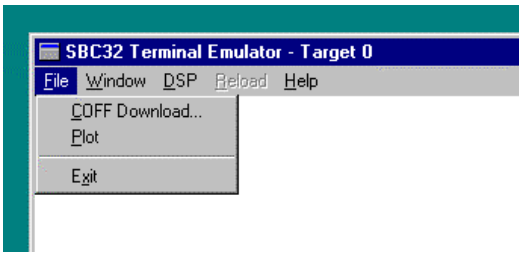


FIGURE 5. Terminal Emulator File Menu

File Menu. File | COFF Download - provides for COFF program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the pathname to the COFF filename to be downloaded is selected by the user. Clicking “Open” in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking “Cancel” will abort the file selection and close the requester with no download taking place.

Q62 Users: Terminal supports downloading of `.OUT` or multi-processor `.MPO` files. `.MPO` files provide a means of downloading separate `.OUT` files to multiple processors simultaneously, which greatly simplifies the task of synchronizing execution in a multi-processor environment.

NOTE: File | COFF Download physically resets the target DSP (in order to initiate the target Talker program) prior to the download. When using the terminal emulator in conjunction with the Code Composer debugger, use Code Composers File | Load Program facility to download executable code to the target rather than the terminal emulator's download facility, since the Code Composer mechanism does not physically reset the target during the download, it is not reliant on the target Talker to perform the download.

FIGURE 6. Diagnostic Received when Target DSP is Halted.

If you attempt to download using the COFF Download menu within the terminal emulator while using Code Composer, you may receive the diagnostic dialog box, which indicates that Code Composer has *halted* the target processor via the JTAG hardware link. While in this halted state, the terminal emulator cannot invoke the Talker program on the target DSP in order to perform the software download. To correct this problem, execute the Debug | Run Free menu command from within Code Composer to release the DSP from JTAG control. Afterwards, clear the terminal emulator error message dialogs and retry the terminal emulator COFF Download.

File | Plot – opens the Plot dialog box, similar to the one listed below.

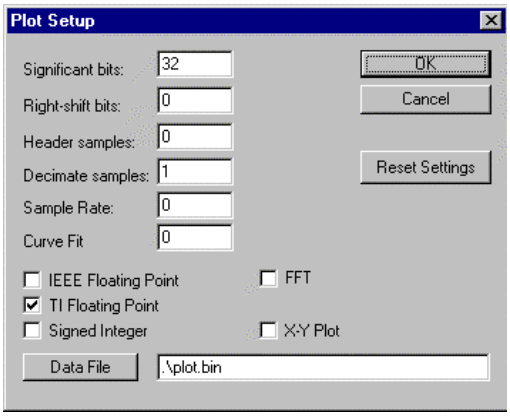


FIGURE 7. Terminal Emulator Plot Menu Dialog Box.

The Plot dialog specifies all of the available options for plotting binary data in Host PC files. Binary data files, usually created by target DSP programs using the `fopen()` and `fwrite()` functions, may contain data in a wide variety of formats which may be plotted in a window from within this dialog box.

Each time data is plotted in the plot window, statistics on the plotted graph are calculated. These statistics are reported in the graph window. The statistics include:

Min displays the minimum value in the data set.

Max displays the maximum value in the data set.

Delta displays the difference between the minimum and maximum values in the data set.

Sdev displays the standard deviation of the data set.

Mean displays the mean value of the data set.

The terminal emulator is capable of plotting files in which binary data has been stored in a wide variety of formats. The default data file format is successive 32-bit (four-byte) values each representing a single TI floating point Y amplitude value. X axis data is not contained in the file and the Y axis amplitude data is plotted against an implied X axis of successively incrementing sample # values, starting at zero.

Each of the available plot options is detailed below.

Edit Boxes . *Significant Bits* specifies the number of significant bits in each data value stored in the data file. The number of bits may range from one to thirty-one. This parameter allows you to plot data gathered from a device at virtually any resolution. For example, if data is accumulated from a 12-bit A/D converter and stored into a binary data file from the target DSP, it would be stored on disk as 16-bit byte-pairs. When plotting this data, with significant bits set to 12, the fallow upper four bits of each 16-bit sample in the data file will be ignored during the data plotting operation.

This parameter indirectly specifies the size of each data sample within the data file, as well. The size of each sample (in bytes) is given by the equation:

Sample size = (significant bits + 7) / 8

The sample size is always the truncated integer result of this formula. Use of the term *sample* throughout the rest of this section refers to clusters of bytes within the data file of size *sample size*.

Shifted specifies the number of bits to shift each data sample stored in the data file, prior to plotting. The number of bits may range from negative thirty-one to positive thirty-one. This parameter allows you to plot data gathered from a device when the output lines of the device are not mapped onto the low-order lines of the data bus. For example, on some of Innovative Integration's DSP boards, a 12-bit A/D is mapped onto data bus bits 15 though 4 rather than on bits 11 through 0. If this data were plotted without modification, the data would erroneously range from –32767 to +32768 rather than the actual 12-bit A/D range of –2047 to + 2048. By specifying a *Shifted* parameter of 4, each data sample extracted from the data file would be right-shifted four bits prior to plotting to compensate for this effect.

Decimate specifies the number of file data points to be skipped between plotted data samples. This option is useful when dealing with a data file containing more than one sample set or in instances where more data is contained in the file than need be plotted. This field must contain a value greater than or equal to one. A value of one specifies that no data should be skipped; a value of two specifies that every other data sample should be discarded, etc.

Header specifies the number of file data samples to be skipped at the beginning of the data file before extracting data to be plotted. This option is used to skip irrelevant data appearing at the beginning of a data file.

Note: Combinations of *Decimate* and *Header* can be used to view individual, 16-bit channels of data acquired as 32-bit pairs on certain DSP boards. For example, the PC31 features two A/D channels, A and B. The A channel is mapped onto the upper 16-bits of the 32-bit data bus while the B channel is mapped to the lower 16-bits of the bus. If this data were written to a data file as 32-bit data, The *Decimate* parameter could be set to 2 to allow plotting of every other sample in the file (all of the A channel data). Further, the *Header* parameter could be set to 1 in conjunction with the above *Decimate* setting to allow skipping of the first sample in the file in order to plot of all of the B channel data in the file.

Fit specifies that the plotted data should be curvefit to the specified order, ranging from zero to five, using a least-squares regression technique. The curvefit data is plotted atop the actual data in red. The correlation coefficient of the fit and the curvefit equation are displayed in the graph window whenever this parameter is greater than zero.

Data File indicates that name of the file containing the data to be plotted.

Radio Buttons. *IEEE* – When checked, indicates that each sample in the data file is stored in 32-bit IEEE-754 floating-point format. When enabled, the Significant Bits and Shifted fields are ignored.

TI – When checked, indicates that each sample in the data file is stored in 32-bit TMS320 TI native floating-point format. When enabled, the Significant Bits and Shifted fields are ignored. This is the default data mode.

Signed – When checked, indicates that each data sample in the data file is signed integer data. When enabled, the Significant Bits and Shifted fields are observed.

NOTE: When *IEEE, TI* and *Signed* are unchecked, the data is assumed to have been stored in the data file as *unsigned* integer data.

XY – When checked, indicates that data samples have been stored in the data file as X-Y (distance, amplitude) pairs rather than in the default data format. In the default format, only the Y (amplitude) data is stored in the file and it is plotted against an implied, incrementing "sample number" X. In the XY mode, samples are parsed from the file and plotted in pairs. Therefore in this mode, half as many points are plotted from the data file.

FFT – When checked, indicates that a Fast Fourier Transform should be applied to the data in the data file prior to plotting.

File | Exit - exits the terminal emulator program.

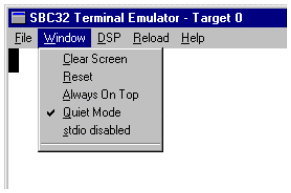


FIGURE 8. Terminal Emulator Window Menu

- Window | Clear Screen - clears the terminal emulation screen and resets the current cursor position to the top left hand corner.

- Window | Reset - causes the terminal emulator to reset all internal stdio processing and clears the screen. If processing is currently halted (via the File | stdio Disabled command), it is then re-enabled. The Reset command is useful when the terminal emulator needs to be initialized prior to running a new DSP application on the target. This can become necessary because the emulator uses multi-character control codes to implement cursor movement and screen control functionality. It is also possible to halt DSP processing (via the JTAG debugger interface) in the middle of a stdio call, which is processing a multi-character sequence. If the program is not continued, this causes the terminal emulator to misinterpret subsequent, new stdio activity. Terminal emulation should always be reset, either via this menu entry or by calling the `stdio_reset()` function within the new application, before new stdio activity is attempted.

- Window | stdio Disabled - a toggling command which allows the user to temporarily disable stdio emulation. This will cause the DSP program to halt at the next stdio library call, and remain paused until stdio processing is again re-enabled by selecting this menu entry. stdio activity processing is halted while the menu entry is checked.

- Window | Always On Top - a toggling command which will cause the terminal emulator to float above other windows on the desktop. This is useful when running stdio-based code from within the Code Composer environment, where the terminal needs to be visible at all times. The terminal will remain atop other windows when this entry is checked. Select the entry again to uncheck and allow the terminal emulator window to be obscured by other windows.

- Window | Quiet Mode – Disables verbose error and diagnostic messages during terminal execution.

DSP Menu

- DSP | Reset - causes the terminal emulator to momentarily assert the target's physical reset pin, bringing the target board into a cold-start, initialized condition.

- DSP | Interrupt - causes the terminal emulator to trigger a target mailbox interrupt using the test code of 0x80 as the signal value. Helpful during testing of target interrupt handlers.

Reload Menu

- Reload - Causes the terminal emulator to re-download and restart the last COFF application previously selected with the File | COFF Download command.

Help Menu

- About - presents program copyright and version information plus information pertaining to the use of Host resources by the target DSP board.

Terminal Emulator Command Line Switches. The terminal emulator also provides the following command line switches to further modify program behavior. The switches must be supplied via the command line or within Windows shortcut properties (see the Installation section for more information), and will override the default behavior of the applet.

- **-t***X* - address selector switch, which allows the user to force the terminal emulator to interact with a specified target. This switch is particularly useful in multi-board installations to create instances of the emulator for targets other than target 0. See the Installation section for more information on multi-board installations. The *X* parameter specifies the logical target number with which to communicate. NOTE: For single-board targets, specify target 0 for boards connected via COM1 and target 1 for boards connected via COM2.

- **-f***filename* - address selector switch, which allows the user to force the terminal emulator to download the specified file to the target DSP board, as soon as the terminal emulator is loaded. This switch is particularly useful in situations where the terminal emulator is "shelled to" from within an other Host applications to facilitate the automatic execution of target applications employing standard I/O.

The COFF File Downloader

The COFF downloader utility provides users with the capability to download and execute COFF files generated by the C compiler or Hypersignal toolsets. This allows users to distribute executable applications independent of the DSP development tools.

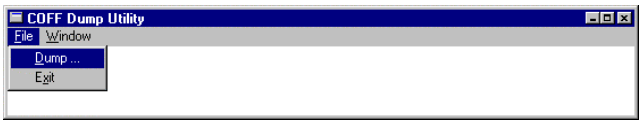


FIGURE 9. The Coff File Downloader Applet

The COFF downloader is simple to use. Double click on the COFF Downloader icon and the program will start and will open a small window with two menu entries, File and Window. To download an application, click on File | Download. This will present a file requester dialog box containing a list of suitable COFF files (`.OUT`) which can be downloaded. Select the desired target executable and click OK to proceed. Click Cancel to abort the download command without selecting a filename.

Once a file is selected, the target will be reset-cycled (to restart its talker), the program will be downloaded and the application launched on the DSP. If any errors are encountered during the download or the download fails to succeed for any reason, an error message box will appear. Typical reasons for failure include improper file selections (a nonexistent or non-COFF format file was selected for download) or errors in hardware or software installation. If repeated errors are noted, proceed to the Installation Troubleshooting section below.

The COFF downloader provides for automated downloads for use in situations where a single application needs to be downloaded and run on the target each time the system is brought up. This can be valuable when placed in the Windows Start-up Folder to automatically download a specific DSP program each time Windows is restarted.

Q62 Users: Download supports downloading of `.OUT` or multi-processor `.MPO` files. `.MPO` files provide a means of downloading separate `.OUT` files to multiple processors simultaneously, which greatly simplifies the task of synchronizing execution in a multi-processor environment.

The File | Exit menu selection will terminate the download application.

COFF File Downloader Menu Commands. The following is a brief description of commands available from the COFF Downloader menus:

File Menu

- File | COFF Download - provides for COFF program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the pathname to the COFF filename to be downloaded is selected by the user. Clicking “Open” in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed.

Clicking “Cancel” will abort the file selection and close the requester with no download taking place.

NOTE: File | COFF Download physically resets the target DSP (in order to initiate the target Talker program) prior to the download. When using the terminal emulator in conjunction with the Code Composer debugger, use Code Composers File | Load Program facility to download executable code to the target rather than the terminal emulators download facility, since the Code Composer mechanism does not physically reset the target during the download and is not reliant on the target Talker to perform the download.

- File | DSP Reset - causes the terminal emulator to momentarily assert the target’s physical reset pin, bringing the target board into a cold-start, initialized condition.

- File | Exit - exits the terminal emulator program.

Window Menu

- Window | Quiet Mode – Disables verbose error and diagnostic messages during terminal execution.

- Window | About - presents program copyright and version information.

Reload Menu

- Reload - Causes the terminal emulator to re-download and restart the last COFF application previously selected with the File | COFF Download command.

COFF File Downloader Command Line Switches. The COFF Downloader also provides the following command line switches to further modify program behavior. These switches must be used in Windows 95/NT shortcut icons (see the Installation section for more information), and will override the same selection made in the configuration utility.

- ***-tX*** - target number selector switch, which allows the user to force the terminal emulator to interact with the specified target. This switch is particularly useful in multi-board installations. See the Installation section for more information on multi-board installations. The X parameter specifies the logical target number with which to communicate. For single-board targets, specify target 0 (zero) for boards connected via com1 and target 1 (one) for boards connected via com2.

- **-q** - force quiet mode switch, which causes the terminal emulator to omit non-fatal warning messages. Fatal errors are still presented in message boxes.

- ***-dpathname*** - cause the downloader to automatically download the named file. Complete path and filename must be given (as in `c:\sbc32cc\hello.out`).

The COFF File Dump Utility

The COFF downloader utility provides users with the capability to generate a report detailing the memory usage of target DSP programs generated using the TI tool set.

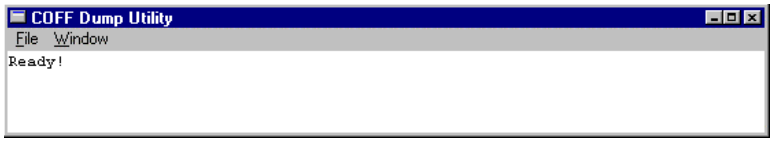


FIGURE 10. The COFF Dump Utility

COFFDUMP.EXE parses through COFF files stored in files on the hard disk and ascertains the complete memory consumption by the DSP program. Memory usage for each of the sections defined in the applications command file are tabularized and the results are written to the Windows NotePad scratch buffer. If desired, Notepad can then be used to write the data to disk or to a printer.

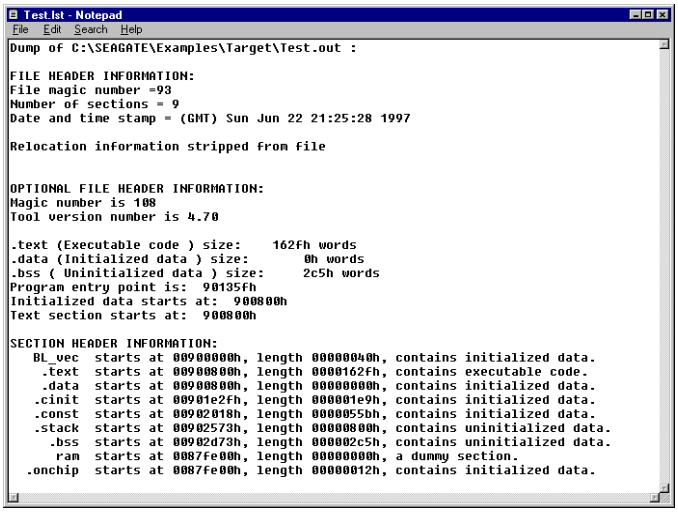


FIGURE 11. COFF Dump Utility Output.

COFF Dump Utility Menu Commands. The following is a brief description of commands available from the COFF Downloader menus:

File Menu

- File | Dump – Involves the standard Windows file selector window for COFF output files (.OUT). Parses through selected file and writes diagnostic dump of contents of executable image to NotePad scratch buffer.

- File | Exit - exits the dump utility program.

Window Menu

- Window | About - presents program copyright and version information.

CHAPTER 5

Developing Target Code

Introduction

The Innovative Integration (I.I.) Zuma Toolset allows users of I.I. DSP processor boards to develop complete executable applications suitable for use on the target platform. The environment suite consists of the TI Optimizing C Compiler, Assembler and Linker, the Code Composer debugger and code authoring environment as well as I.I.'s custom Windows applets (such as the `TERMINAL.EXE` terminal emulator).

Code Composer Studio is the default package used to automate executable build operations within Innovatives Zuma Toolsets, simplifying the edit-compile-test cycle. Source is edited, compiled, and built within Code Composer Studio, then downloaded to the target and tested within either the Code Composer Studio debugger or via the Zuma terminal emulator.

On C6x platforms, such as Innovatives M6x, SBC6x and Quatro6x, Code Composer Studio may be used for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms using Studio are provided in this chapter.

Do not confuse the creation of target applications (code running on the target DSP processor) with the creation of host applications (code running on the host platform). The TI tools generate code for the TI DSP processors, and are a separate toolset from that needed to create applications for the host platform (which would consist of some native compiler for the host processor, such as Microsoft's Visual C++ or Borland Builder C++ for IBM compatibles). To create a completely turn-

key application with custom target and host software, *two* programs must be written for *two* separate compilers. While I.I. supports the use of Microsoft C/C++ for generation of host applications under Windows with sample applications and libraries, we do not supply the host tools as part of the Development Environment. For more information on creating host applications, see the section in this manual on host code development.

This section supplies information on the use of the development environment in creating custom or semicustom target DSP software. It is not intended as a primer on the C language. For information on C language basics, consult one of the C primer books available at your local bookstore. The definitive reference to the C language is The C Programming Language, by B. Kernighan and D. Ritchie (Prentice Hall. Englewood Cliffs, NJ. 1988).

Components of Target Code (.c, .asm, .cmd)

In general, DSP applications written in TI C require at least two files: a `.c` file (or "source" file) containing the C source code for the application, and a `.cmd` file (or "linker command" file) which contains the target-specific build data needed by the linker. There may also be one or more `.asm` assembler source files, if the user has coded any portions of the application in assembly language.

Edit-Compile-Test Cycle using Code Composer Studio

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion.

When using the Code Composer Studio, these stages of the cycle are accomplished entirely within the Studio integrated environment. The project features of Code Composer Studio support the project and component file editing and compilation stages, along with allowing the executable result to be downloaded and tested on the target hardware.

A Simple Code Composer Studio Project

The following sequence illustrates the creation of a project to build the "`Hello World!`" program from within Code Composer Studio.

First, start Code Composer Studio. Select Project | New from the Project menu and you will see the following dialog:

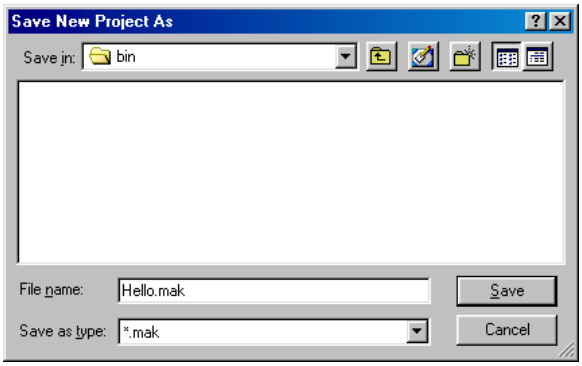


FIGURE 12. Creating a New Project in Code Composer Studio

Browse to the directory in which you would like to create the new project (your working directory) and then type the name of the new project. In this example, the working directory is c:\ti\bin and the project name is hello.mak. In the standard developers package, you may browse into the <I.I. target board>\EXAMPLES\TARGET directory.

Next, open the Project | Add Files to Project dialog box to add files to the project. Add the HELLO.C file from the C:\<I.I. target board>\Examples\Target directory and the GENERIC.CMD file from the C:<I.I. target board> directory to the project. Remember to choose the correct file type then when you have selected the file to add, click **<Open>** button.

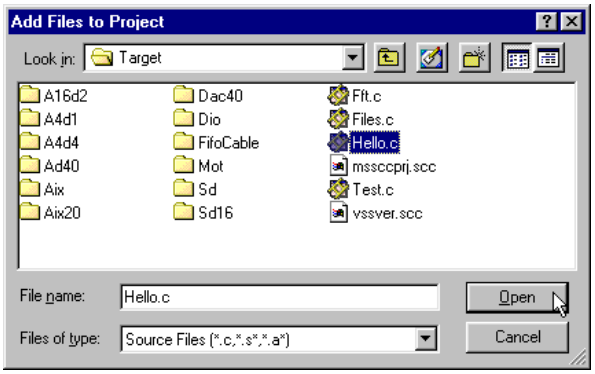


FIGURE 13. Adding Files to a Code Composer Studio Project

It is imperative that you add an appropriate command file to the Code Composer Studio project. The generic.cmd command file describes the memory map of the target hardware, without which the linker will be unable to place executable sections into appropriate memory regions for debugging. That is, the memory map for the target DSP specified in the generic.cmd file will be used to link the project output file. If you wish, you may copy the contents of the generic.cmd file (located in the root of the Zuma toolset) into your working directory, rename it appropriately and add the modified cmd file to your project instead.

The library files will be required, but do not add them directly into the project like the hello.c and generic.cmd files. Rather, manually type the desired libraries needed to link the project into the Project | Options | Linker tab when instructed to do so later within this chapter.

Next, you may optionally open the files in the project by double-clicking on their names within the Project window.

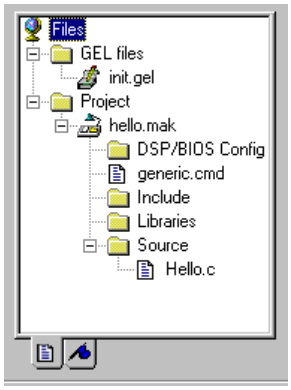


FIGURE 14. Code Composer Studio Project Window.

Next, you must configure the project compiler settings so that when Hello.c is compiled, the appropriate memory model and switches are used.

Build Options (M62, Q62, SBC62 Boards)

Click on Project | Options to open the Build Options dialog box, then click on the Compiler tab to show the current compiler options.

Configure the compiler options to use the following settings: In the Memory Models combination box, select “Aggregate Data and Calls are far (-ml2 memory model)”. Then in the category column, choose Optimization and select “Level 2 - Global”. In the category column, choose Assembly and de-select “Keep Labels as Symbols”. Again in the category column, choose Runtime Model Options 1 and select “Big Endian Format”. Then type the last compiler build option “-x2” in the edit box at the end of the

command line being edited. When finished, the compiler dialog screen should look exactly like one below.

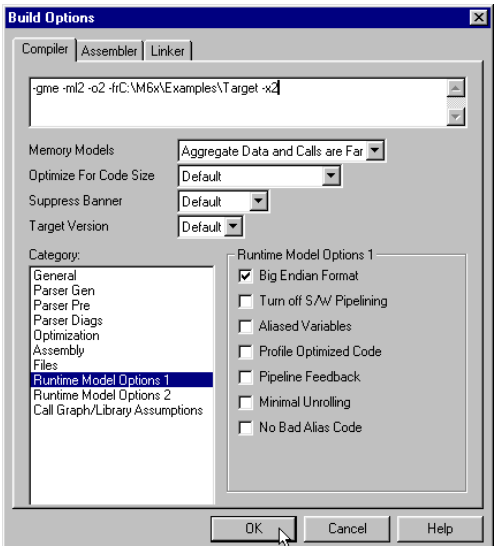


FIGURE 15. Code Composer Studio Compiler Build Options

Next, click on the Assembler tab and configure the assembler build options. In this screen, make sure the following options are selected “Enable Symbolic Debug Information”, “Make All Symbols Global” and “Produce big-endian format codes”. When finished, the assembler dialog screen should look like one below.

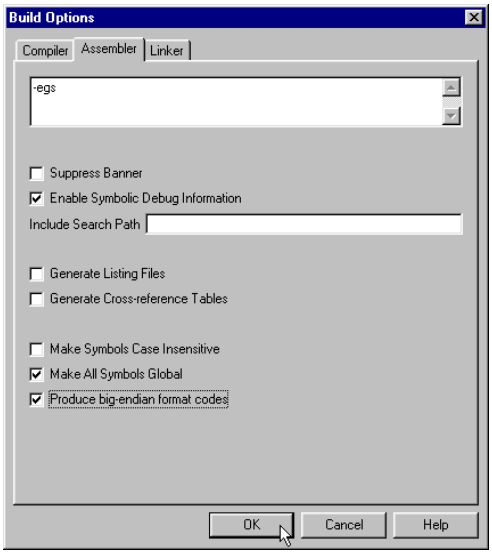


FIGURE 16. Code Composer Studio Assembler Build Options

Finally, click on the Linker tab and configure the linker build options as follows. In the Output Module combination box select “Relocatable Executable”. Then set the Heap Size to 0x400 bytes and the Stack Size to 0x800 bytes. Make sure the Exhaustively Read Libraries has been selected. Now, add stdio.lib; periph.lib; dsp.lib; and rts6201e.lib into the Include Libraries edit box (in that order). When finished, the linker dialog screen should look like one below.

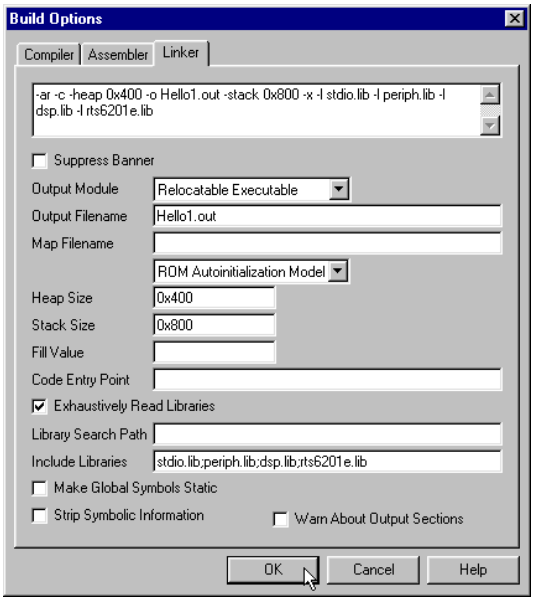


FIGURE 17. Code Composer Studio Linker Build Options

Once all the build options have been set, rebuild your project by clicking Project|Rebuild All in the Code Composer Studio menu bar. If errors are encountered in one or more source files, they are listed in the output window. You may visit and repair each error by either double clicking on each error in the Output window.

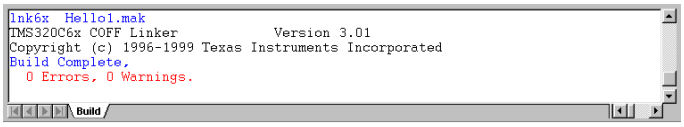


FIGURE 18. Code Composer Studio Build Results Window

Build Options (M67, Q67, SBC67 Boards)

Click on Project | Options to open the Build Options dialog box, then click on the Compiler tab to show the current compiler options.

Configure the compiler options to use the following settings: In the Memory Models combination box, select “Aggregate Data and Calls are far (-ml2 memory model)”. In the Target Version combination box, select “67xx”. Then in the category column, choose Optimization and select “Level 2 - Global”. In the category column, choose Assembly and de-select “Keep Labels as Symbols”. Again in the category column, choose Runtime Model Options 1 and select “Big Endian Format”. Then type the last compiler build option “-x2” in the edit box at the end of the command line being edited. When finished, the compiler dialog screen should look exactly like one below.

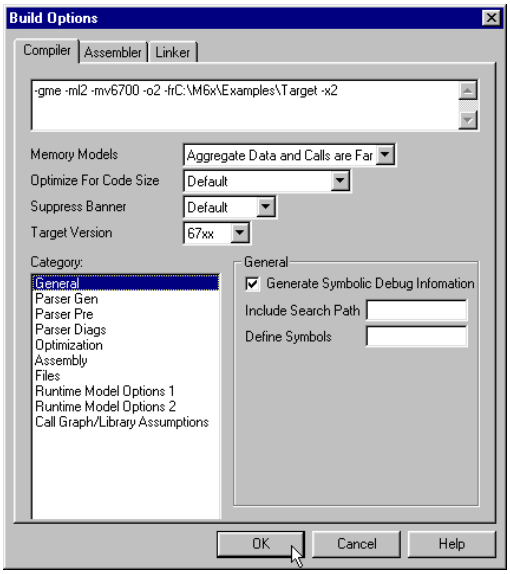


FIGURE 19. Code Composer Studio Compiler Build Options

Next, click on the Assembler tab and configure the assembler build options. In this screen, make sure the following options are selected “Enable Symbolic Debug Information”, “Make All Symbols Global” and “Produce big-endian format codes”. When finished, the assembler dialog screen should look like one below.

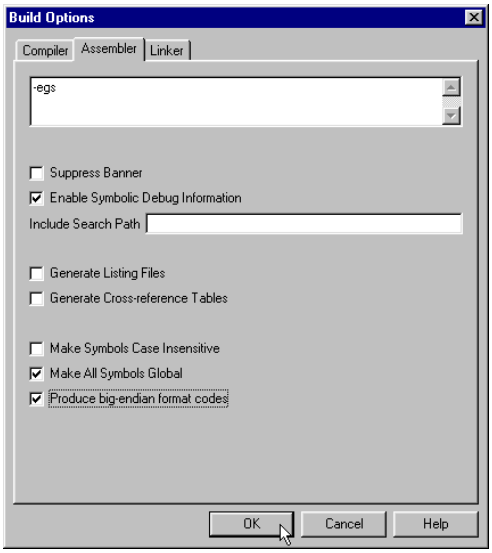


FIGURE 20. Code Composer Studio Assembler Build Options

Finally, click on the Linker tab and configure the linker build options as follows. In the Output Module combination box select “Relocatable Executable”. Then set the Heap Size to 0x400 bytes and the Stack Size to 0x800 bytes. Make sure the Exhaustively Read Libraries has been selected. Now, add stdio.lib; periph.lib; dsp.lib; and rts6701e.lib into the Include Libraries edit box (in that order). When finished, the linker dialog screen should look like one below.

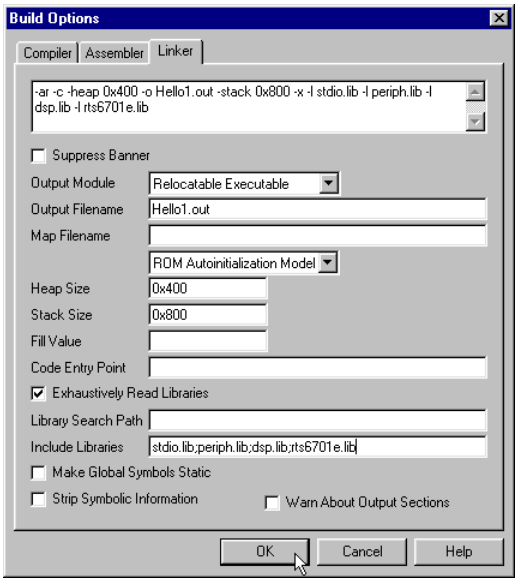


FIGURE 21. Code Composer Studio Linker Build Options

Once all the build options have been set, rebuild your project by clicking Project|Rebuild All in the Code Composer Studio menu bar. If errors are encountered in one or more source files, they are listed in the output window. You may visit and repair each error by either double clicking on each error in the Output window.

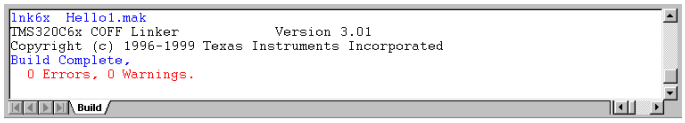


FIGURE 22. Code Composer Studio Build Results Window

Automatic makefile creation

When a project is created, opened, modified, built or rebuilt, the Code Composer Studio dependency generator automatically generates a project makefile (named `<project file>.mak`, located in the project directory), which is capable of rebuilding the project's output file from its components.

This file is automatically submitted to the internal make facility whenever you click on build or rebuild within Code Composer Studio. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

Rebuilding a Project

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project. To force a project rebuild, select Project | Rebuild All from the Code Composer Studio menu bar.

Running the Target Executable

The `hello` program is very simple, only printing the single line "Hello, World" to the terminal emulator before waiting to echo any keystrokes and exiting. Bring up the "Hello, World" source file edit screen. Scroll down the source file by using cursor down button until you reach the call to printf(), which looks like the following:

```
printf("Hello, World\n");
```

Change the output string to read "`Hello, Brave New World\n`". You can now compile the new version by executing Build from the Project menu (or by clicking on its toolbar icon). This causes Code Composer Studio to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`hello.obj`). Code Composer Studio then invokes the TI Linker using the `generic.cmd` file, which is located in the root

board directory. This rebuilds the executable file using the newly revised `hello.obj` . If no errors were encountered, this process creates the downloadable COFF file `hello.out`, which can be run on the target board. At this point, the program may be run using the Terminal Emulator applet, which may be invoked using the Terminal shortcut located within the target board program group created during the Zuma Libraries installation process. In the terminal emulator, download the hello.out file. The program runs and outputs the message “Hello, Brave New World” to the terminal emulator window.

If errors are encountered in the process, Code Composer Studio detects them and places them in the build output window. If the error occurred in the compiler or assembler (as in a C syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the build output window, and the error message will be displayed in the Code Composer Studio status bar. If the linker returns a build error, the build output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is misspelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the build output window.

Note: Be sure to start the terminal emulator **BEFORE** starting Studio, to avoid resetting the DSP target in the midst of the debugging session. If Terminal is not yet running and you wish to run the Hello object file, perform the following steps.

1. Execute Debug | Run Free to logically disconnect the DSP from the debugger software.
2. Terminate the Studio application.
3. Invoke the Terminal application.
4. Restart the Studio application.

This outlines the basics of how to recompile the existing sample programs within the Studio environment.

Anatomy of a Target Program

While not providing much in the way of functionality, the `hello` program does demonstrate the code sequence necessary to properly initialization the target. The exact coding, however, is very specific to the I.I. C Development Environment, target boards, and is explained in this section in order to acquaint developers with the basic syntax of a typical application program.

Here we examine the M62 version of the `hello` program example. Although the source is not necessarily identical to that of `hello` for the other targets, it is typical of the overall structure of the typical application program designed under the development environment.

```
/*
*       HELLO.C
*       Test file/program for target board.
*/

#include "periph.h"
#include "stdio.h"

main()
{
        int key;
        enable_monitor();
        clrscr ();

        printf("Hello World!\n");
        printf("\nEchoing keystrokes...\n");

        do
                {
                key = getchar();
                putchar(key);
                }
        while(key != ESC);

        monitor();
}
```

The two lines of the program that being with a "#" are #include statements, which include the header files for the peripheral and standard I/O libraries. These include prototypes for all the library routines as well as variable definitions and #define statements for the peripheral memory-mapping addresses. These #defines are especially important for those who wish to perform direct peripheral access, rather than using the peripheral libraries.

The `enable_monitor()` will setup the standard monitor I/O interface. The next two lines perform the standard I/O function of the program, clearing the terminal emulation screen and printing "Hello, World" & "Echoing Keystrokes...". These two lines are where custom code should be inserted.

The following `getchar()` call simply echoes keys typed at the terminal emulator back to the terminal display. This routine is also part of the standard I/O library. The program effectively terminates here, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

The `hello` program is very simple, but it exhibits the basic functionality needed to properly start on the CPU, as well as the initialization needed to interact with `Code Composer Studio` and the terminal emulator properly in the development environment.

Use of Library Code

Library routines can be compiled and linked into your custom software simply by making the appropriate call in the source and adding the appropriate library to the linker command file. Refer to the library reference in this manual for library location information on each function.

In general, user software needs to #include the relevant library header file in source code. The header files define prototypes for all library functions as well as definitions for various data structures used by the library functions. The file stdio.h should be included by programs using the standard I/O library, and the file periph.h should be included if a program uses functions in the peripheral library. The function definitions in the peripheral library reference note which library a particular function lives in, as well as the header file, which should be included for that function.

Compiling/Assembling/Linking Outside Code Composer Studio

Under certain circumstances, it may not be possible to use Code Composer Studio macro definitions to compile inside the editor. `COMPILE.BAT`, `ASSEMBLE.BAT`, and `LINK.BAT` are provided in the %II_BOARD% directory and may be executed by typing their names followed by the source file on which they are to operate. For example, the file `mycode.c` can be compiled by typing

```
compile mycode
```

at the DOS prompt. This causes the `COMPILE.BAT` script to start, which runs the compiler and generates the file `mycode.obj`, assuming no errors occurred. The `COMPILE.BAT` script also searches for the file `mycode.cmd` in the current directory. If the linker command file is found, then the linker is automatically run and the entire executable linked. If the command file is not found, processing stops with the generation of `mycode.obj`.

Assembly source (`mycode.asm`) may be assembled by typing

```
assemble mycode
```

where the assembler is called and an object file generated.

Linking can also be performed. In this case the input file is not source code, but a linker command file (`mycode.cmd`):

```
link mycode
```

This line causes the linker to build the executable `mycode.out`, again assuming no errors have occurred during the process. Also, note that the `COMPILE.BAT` script will automatically link the executable if a linker command file of the same name exists.

In all the above cases, if any errors occur, an error file (mycode.err) is generated by the software tools. The mycode.err file contains the full console output of each of the tools. Any error that is generated by the tools will be recorded in this file.

The Next Step: Developing Custom Code

In building custom code for an application, I.I. recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

CHAPTER 6

Developing Host Code

This section describes the Innovative Integration Windows host software development environment. The environment provides complete support for generating 32-bit Windows-compatible software, which is capable of controlling and communicating with Innovative Integration's DSP co-processor and data acquisition cards. Virtual device drivers (Windows 9x VxD or NT Kernel Mode Driver) and dynamic link libraries (DLL) are included to provide an easy-to-use, portable low-level interface for the target hardware. Sample applications show how to call the DLL functionality and present basic interface examples with guidelines for on processor card control requirements and data movement.

Host software development is directly supported under the Microsoft Visual C/C++ 4.0 environment for generating 32-bit Windows applications. All example application programs included in the development package are supplied with Visual C workspace files, making program modification and regeneration as simple as possible.

Please Note: Only Windows application development is currently supported by the Developer's Package. Foreign operating systems, such as Unix and OS9 are not currently supported.

Dynamic Link Library

All target interactions takes place through calls to the supplied dynamic link library (DLL). This library supplies low-level functions for basic target board control, including processor reset/run state, message passing via the board-specific mailbox registers, application downloading, and bus master memory locking and access control.

The function calls available under the DLL are documented in the appendices. Sample applications (described below) provide working examples on how to interact with the card via host software.

Sample Host Programs

The DLL is capable of interacting with up to four target DSP boards simultaneously by default (contact II if more than four targets are required). The DLL maintains a board-specific structure of information for each target, known as the `cardinfo` structure. An prototype of the cardinfo structure is located in the `\INCLUDE\HOST\` subdirectory in the `CARDINFO.H` file. An example is shown below.

```
//
//    cardinfo.h   --   definition of CARDINFO structure
//

#ifndef __CARDINFO_H__
#define __CARDINFO_H__

#include "ii_iostr.h"       // Common IO Driver/DLL Structures
#include "mailbox.h"        // Definition of MAILBOX structures

//
//  BoardInfo structure
//
typedef struct _BoardInfo
{
    ULONG        ProcessorCount;
    ULONG        DLL_Version;      // Version ID numbers
    ULONG        DrvVersion;
    ULONG        TalkerVersion;
    ULONG        CellSize;         // Target memory cell size, in bytes
    ULONG        CtlReg;           // Shadow of control register
    ULONG       FlashSectorSize; // Size of flash sectors, in bytes
    ULONG       FlashDeviceId;   // Flash device ID
    ULONG        QuietMode;        // Don't Display Messages if true
} BoardInfo;

//
//  InterruptInfo structure
//
typedef struct _InterruptInfo
{
    ULONG        IRQ;              // IRQ of attached interrupt
    HANDLE       Ring0Event;       // Ring 0 event handle
    HANDLE       Ring3Event;       // Ring 3 event handle
    void         (*Vector)(void *); // Virtual ISR function pointer
    void *      Context;         // Virtual ISR context pointer
 } InterruptInfo;
```

```
//
//  SerialInfo structure
//
typedef struct _SerialInfo
{
  LONG           In;         // Buffer for last character received
  LONG           ReadFlag;   // True when character received
  LONG           MbValue;    // Multi-byte value
  LONG           MbCtr;      // Multi-byte read state
  ULONG          RTS_state;  // Current state of the RTS output
  LONG          Bcr;    // Bus control register value for Flash access
  LONG           Reading;    // TRUE if currently reading a character
  OVERLAPPED     RxOverlap;  // Info used in asynch input
  OVERLAPPED     TxOverlap;  // Info used in asynch output
  COMMTIMEOUTS   Timeouts;   // Info for set/query time-out parameters
  DCB            Dcb;        // Device control block
} SerialInfo;


//
//  CARDINFO  structure
//
typedef struct _cardinfo
{
   ULONG           Target;         // Number of current target
   HANDLE          Device;         // Handle to Driver for device
   BoardInfo       Info;           // Board Info
   MAILBOX *       Mail;           // Talker Mailbox Array
   IoPortBlock     Port;           // Primary Port Block Information
   IoPortBlock     OpReg;          // Secondary Port Block Information
   MemoryBlock     DualPort;       // Shared Memory Area Information
   MemoryBlock     BusMaster;      // BusMaster Memory Information
   nterruptInfo    Interrupt;      // Interrupt Information
   SerialInfo      Serial;         // Serial Port I/O (SBC's)
} CARDINFO;

#endif
```

The `cardinfo` structure is accessed within Host application programs in order to gain access to board-specific parameters which are maintained by the DLL. For example, in order to ascertain the size of the shared memory area on a specific target card a host program could use:

```
/* send bus mastering physical address to target processor */

dsp = (CARDINFO*)target_cardinfo(target);

size = dsp->Dualport.Size;
```

Sample Host Programs

Each Zuma Toolset is supplied with one or more example programs which illustrates control of the DSP board via the supplied DLL. For bus-based boards, the example is SCOPE.C, which emulates a simple oscilloscope. For stand-alone boards, the XRPT.C example is provided, which illustrates advanced serial communications. The SCOPE example

`SCOPE.C` is a small, working example written in Visual C v4.0 showing how to use bus-based DSP boards to move data between the target and Host memory spaces. The host application works in concert with a small DSP program running on the target to mimic the operation of a simple oscilloscope.

The SCOPE application is included in the `\EXAMPLES\HOST\SCOPE` subdirectory of `%II_BOARD%`. Its executable is located in the `\EXAMPLES\HOST\SCOPE\RELEASE` subdirectory. The DSP support code for this application is located in the `\EXAMPLES\HOST\SCOPE\DSP` subdirectory.

SCOPE.C is a multi-threaded application example with three threads. The primary thread performs Window management, including the Windows message handler. A second thread, `EnqueueData()` handles data movement from the target DSP to the Host using shared memory (dual port memory on ISA bus cards and bus master memory on PCI cards). The third thread, `PlotData()`, plots the enqueued data received from the target within the window.

This program illustrates many of the elements of a typical Host application, which communicates with a target DSP application. In this example, the Host program communicates closely with the `SCOPE.C` DSP application located in the `\EXAMPLES\HOST\SCOPE\DSP` directory. SCOPETRG.C is the code which runs on the target DSP and is responsible for feeding information to the Host via shared memory.

When the Host program starts, it invokes the COFF downloader to download the object image of the target DSP application (SCOPETRG.OUT) within the `download()` procedure. This procedure makes calls on the DLL in order to effect the download. Following the download, the application is started running using the `start_app()` function. The DSP application immediately begins generating mock analog data in order to emulate acquiring data from the analog subsection on the target, enqueuing the acquired data. As soon as a packet-full of data is available, the data is dequeued by the target, moved into shared memory and the Host program is signalled, using the `host_interrupt()` target procedure.

The Host device driver handles the target interrupt signal and issues an special EVENT message to the ring three DLL which performs a callback on the user-installed Host `EnqueueData()` function. When this occurs, the offset into dual port memory containing the new packet of analog samples is read from the shared memory. This address is used to enqueue data from shared memory area into a Host-maintained data queue of real-time analog samples.

The Host `PlotData()` thread draws an oscilloscope-like grid on the display window, then polls continuously for the availability of analog samples in the Host queue. When a screenfull of data is available in the queue, it is dequeued and plotted.

The primary thread is responsible for handling window messages only. The most typical window messages are invoked when the user drags or resizes the oscilloscope window. When this occurs, the `WM_SIZE` message handler sets the global variable refresh TRUE, which indicates to the `PlotData()` thread that a complete window update is needed. The `PlotData()` function temporarily drops out of the data plotting loop in order to redraw the oscilloscope display. Then, it resumes the plotting function again, until the refresh variable is modified again.

The XRPT Example

`XRPT.C` is a small, working example written in Visual C v4.0 showing how to use serial-based DSP boards to move data between the target and Host memory spaces. The host application works in concert with a small DSP program running on the target to tally the number of target-to-host interrupts signalled by the DSP during application execution. Like the SCOPE example for bus-based DSPs, XRPT illustrates installation of a Host interrupt handler using DLL calls. This interrupt handler is invoked by the target DSP via the host_interrupt() function call, which in the case of single-board targets initiates a delta-CTS interrupt to the Windows device driver, which signals and event to the II DLL which calls back the user-installed interrupt function.

CHAPTER 7

Creating Target Software

Software is created for the target DSP by using one or more of the tools included in the Developer's Package. The tools can be used alone or in concert with each other to generate a downloadable executable COFF format file, which can be run on the target DSP board with the aid of the utilities included in the developer's package.

This section of the *Developer's Package Manual* details the use of the individual tools in the package to create executables for the target DSP. This section also gives step-by-step instructions on how to use the C compiler and Code Composer to write, compile, test, and debug custom C applications on the target. Sample C applications are also discussed

C Code Development

C Compiler

The Texas Instruments C compiler is an ANSI C compatible compiler, which produces optimized assembly code for the TMS320C4x family of processors. A complete set of manuals is included with the M62 Developers Package.

In addition to the excellent manuals from TI, refer to the Kernighan and Ritchie C Handbook (available at cost from I.I.) for generic C questions and syntax. The TI manuals primarily describe the use of the compiler with the TMS320C4x family and are not intended as C primers for the beginner.

C Library Reference

Complete source code to the entire suite of ANSI C libraries is provided with the C system to aid in code development. Refer to the *TMS320 Floating Point DSP Optimizing C Compiler Manual* for a complete list of TI C functions.

The I.I. M62 Developer's System also includes extensive high-level libraries useful in interacting with the various peripherals on the M62 board. The following sections describe by peripheral type the functions provided in the peripheral library. For a complete alphabetical listing of all peripheral functions, see Appendix.

M62 Zuma Toolset Libraries

The Zuma toolset provides both target peripheral libraries and Host DLLs along with numerous example programs to illustrate usage.

The peripheral libraries for the M62 provide support for the on-board peripherals and terminal I/O functions. The libraries are provided in three linkable `.LIB` files: `PERIPH.LIB`, `STDIO.LIB, and DSP.LIB`. `STDIO.LIB` holds all the console terminal emulation and communications routines listed in the following section, while `PERIPH.LIB` contains all other peripheral driver routines. DSP.LIB contains commonly requested C-callable digital signal processing functions, plus common math and queue management extensions. Source code for the routines is also provided, arranged by function in the `\PERIPH, \STDIO`, and `\DSP` subdirectories of the root II_BOARD directory, as follows:

Directory	Library Source
`\DSP`	Standard Digital Signal Processing Routines.
`\PERIPH\ANALOG`	Drivers for the M62 A4D4 instrumentation-grade analog I/O module and the complementary TERM mux module. Drivers for the SD high-performance audio module.
`\PERIPHERAL\BUS`	Drivers for V360 bus-mastering PCI interface.
`\PERIPH\DIGITAL`	Digital I/O, PIT Timer control, module FLASH ROMs, etc. Drivers for DIO module. Drivers for MOT motion control module.
`\PERIPHERAL\DIO`	DIO module DUART and digital I/O drivers.
`\PERIPH\MISC`	Miscellaneous processor control and data conversion functions.
`\PERIPH\RTS`	Modified boot-up routines for the M62 baseboard.
`\STDIO`	Console and terminal emulation functions.
`\TALKER`	Start-up umbilical 'C6201 software.

TABLE 5. Zuma Toolset Source Directories

The toolset also contains various support files arranged as described below.

Directory	Library Source
\EXAMPLES\HOST	Example programs illustrating use of the DLL to control the DSP board from within MS Visual C programs.
\EXAMPLES\TARGET	Example programs illustrating use of the target peripheral libraries to perform common DSP tasks.
\INCLUDE\HOST	Header files used by Host Visual C programs.
\INCLUDE\TARGET	Header files used by target Texas Instruments C programs.
\LIB\HOST	Linkable library files for Host Visual C and C++ programs.
\LIB\TARGET	Linkable library files for target TI C and assembler programs.
\SRC	Useful public domain source files for the C6201 processor.

TABLE 6. Zuma Toolset Support Subdirectories

STDIO Console Terminal Driver. The Developer's Package contains a full-featured terminal emulator application (terminal.exe), suitable for both user interface purposes as well as debugging use. The peripheral library provides a complete set of standard I/O routines, which can communicate directly with this terminal emulator. The source for the standard I/O routines is given in the `\STDIO` subdirectory under the installation directory. In general, the standard I/O library functionality is identical to that of the K&R standard I/O library. However, some M62-specific functions are provided to allow higher level functionality such as cursor positioning, text attribute control, and graphical data plotting. The following target programming section gives details on how to use the standard I/O peripheral library to interact with the terminal emulator.

Digital Peripheral Drivers. The digital peripheral drivers control the 'C6201 internal timers and the digital I/O lines. These drivers allow for high-level access to timebase control functions and digital I/O activity without doing direct hardware programming. The following target programming section gives details on how to use the digital peripheral library to program the digital peripherals. Source code for the functions is given in the `\PERIPH\DIGITAL` directory.

BUS Peripheral Drivers. The BUS peripheral drivers provide control functions for the onboard V360 PCI bus interface. The available routines support very-high speed bus-mastering transfers between the 512 Kbyte, external async SRAM of the M62 and host PC memory. This driver also includes hardware mailbox support routines, which are used extensively by the standard I/O library in order to support terminal emulation. Additionally, these mailbox routines provide a means of performing interrupt-driven communications with the Host PC. The target programming section gives details on how to use the bus peripheral library. Source code for the functions is given in the `\PERIPH\BUS` directory.

Miscellaneous Peripheral Drivers. The MISC directory contains code to support high-level access to the internal registers, byte packing and unpacking, interrupt vector support, and other functions. Source code for the functions is given in the `\PERIPH\MISC` directory.

RTS Peripheral Drivers. The RTS peripheral drivers provide board-specific versions of the functions called by the TI C Compiler during coldstart initialization of the C runtime engine. These files have been modified as necessary in order to provide a complete initialization of the M62 onboard hardware immediately prior to calling `main()` within application code. Additionally, the RTS functions include a modified version of the millisecond timer function required to support the TI C timekeeping functions (listed in `time.h`). Source code for the functions is given in the `\PERIPH\BUS` directory.

Digital Peripheral Drivers. The digital drivers support access to all baseboard and add-on digital I/O functions.

The DIO peripheral drivers provide control functions for the optional DIO plug-in module. The functions provide high-level C access to the DIO module's 32, additional digital I/O lines, plus either interrupt-driven or polled use of the DIO's onboard DUART (Dual-channel Universal Asynchronous Receiver Transmitter). The target programming section gives details on how to use the digital peripheral library to program the digital peripherals. Source code for the functions is given in the `\PERIPH\DIGITAL\DIO` directory.

The MOT peripheral drivers provide control functions for the optional MOT plug-in module. The functions provide high-level C access to the MOT module's four, precision motion-control axes. Each of the axes features independent encoder inputs and either digital or 16-bit analog output. Digital output may be either pulse and direction positive or negative pulse to support stepper motor amplifier inputs. The target programming section gives details on how to use the MOT peripheral library to program these peripherals. Source code for the functions is given in the `\PERIPH\DIGITAL\MOT` directory.

Analog Peripheral Drivers. The Analog peripheral drivers provide control functions for the optional analog plug-in modules: A4D4, AIX, and SD modules. The functions provide high-level C access to the A4D4's analog input and output channels and their associated gain amplifiers. Additionally, the driver supports control of the optional TERM break-out panel, a companion to the A4D4 module. In order to support muxing of each of the A4D4 modules 8:1 to allow input from up to 32 simultaneous channels per A4D4 module. Source code for the functions is given in the `\PERIPH\ANALOG\A4D4` directory.

The AIX peripheral drivers provide control functions for the optional AIX plug-in module. The functions provide high-level C access to the AIX module's four, 2.5 MHz, 16-bit analog input channels. Source code for the functions is given in the `\PERIPH\ANALOG\AIX` directory.

The SD peripheral drivers provide control functions for the optional SD plug-in module. The functions provide high-level C access to the A4D4 module's four, audio-grade, 24-bit analog input and 20-bit output channels. Source code for the functions is given in the `\PERIPH\ANALOG\SD` directory.

The target programming section gives details on how to use the analog peripheral library to program these analog peripherals.

Digital Signal Processing Library. The DSP directory contains code to support high-level access to the common signal processing functions such as FFT's, filters and compression. Additional routines are provided for common functions such as matrix manipulation, curve fitting and general purpose queue management. Source code for the functions is given in the `\DSP` directory.

Texas Instruments C Libraries. Several libraries are included with the system that provide support for floating point and extended math functions, DSP oriented procedures and initialization examples. Chapter 5 in the *TMS320 Floating Point DSP Optimizing C Compiler User's Guide* describes the libraries.

The following libraries are available:

Library	Operation
ASSERT.H	Defines the assert macro for runtime error message reporting.
CTYPE.H	Declares functions that test and convert characters.
LIMITS.H	Defines range limits for characters and variable types.
FLOAT.H	Defines floating point range limits.
MATH.H	Defines trigonometric, exponential and hyperbolic math functions.
ERRNO.H	Defines errno variable for catching range errors in function calls.
STDARG.H	Defines macros to aid in variable argument functions.
STDDEF.H	Defines two new types and macros used within runtime functions.
STDLIB.H	Declares many common library functions such as string conversion, sorting and searching functions, program exit functions and some integer-arithmetic that is not a standard part of C.
STRING.H	Declares functions for string manipulations.
TIME.H	Declares macros and types useful for time manipulations.

TABLE 7. Texas Instruments Standard Library Functions

M62 Hardware Interaction

All peripherals are memory mapped into the ‘C6201 address space, using the locations given in the following table. The table also lists the wait states applied to accesses to each peripheral.

The development system provides routines to access all integrated M62 peripherals. This section describes how to program the peripherals using the supplied library functions under C or via direct memory accesses to the supplied peripheral register map. In general, direct memory access delivers higher performance than using the C function library since it avoids the overhead of the function calls necessary to access the library. However, the libraries have been crafted to utilize inline code where possible to mitigate this effect. In the peripheral descriptions that follow, each device’s access methods are called out for both high level and direct memory access. In the case of C functions, the function names and argument variables are called out. In the case of direct memory access operations, the relevant addresses are listed along with the functions they perform and accompanying `Periph` structure elements which may be used from C to simplify access. These elements are defined in the header file `periph.h`.

Function	Address	C Language Mneumonic	Mem Space
FIFO Port	0x0400000	Periph->Fifo	CE0
V360 Registers	0x1400000	Periph->PciRegs	CE1
FIFO Port Reset	0x1410000	Periph->FifoReset	
AD9850 Reset	0x1470000	Periph->DDS.Reset	
AD9850 Frequency Update	0x1480000	Periph->DDS.Update	
AD9850 Write Clock	0x1490000	Periph->DDS.Clock	
Digital I/O Data Register	0x14A0000	Periph->Dio.Data	
Digital I/O Direction Control	0x14B0000	Periph->Dio.Direction	
Digital I/O Input Latch Clock Control Register	0x14C0000	Periph->Dio.LatchControl	
External Mux Control 0	0x14D0000	Periph->Mux[0]	
External Mux Control 1	0x14E0000	Periph->Mux[1]	
16 bit External Timer	0x14F0000	Periph->Timer	
External Interrupt Input 4 Select	0x1500000	Periph->EI[4]	
External Interrupt Input 5 Select	0x1510000	Periph->EI[5]	
External Interrupt Input 6 Select	0x1520000	Periph->EI[6]	
External Interrupt Input 7 Select	0x1530000	Periph->EI[7]	
I/O Module Strobe 0	0x1540000	Periph->Module[0]	
I/O Module Strobe 1	0x1550000	Periph->Module[1]	
I/O Module Strobe 2	0x1560000	Periph->Module[2]	
I/O Module Strobe 3	0x1570000	Periph->Module[3]	
I/O Module Strobe 4	0x1580000	Periph->Module[4]	
I/O Module Strobe 5	0x1590000	Periph->Module[5]	
I/O Module Strobe 6	0x15A0000	Periph->Module[6]	
I/O Module Strobe 7	0x15B0000	Periph->Module[7]	
I/O Module Strobe 8 (cM62 only)	0x15C0000	Periph->Module[8]	
I/O Module Strobe 9 (cM62 only)	0x15D0000	Periph->Module[9]	
I/O Module Strobe 10 (cM62 only)	0x15E0000	Periph->Module[10]	
I/O Module Strobe 11 (cM62 only)	0x15F0000	Periph->Module[11]	
Async SRAM (128Kx32)	0x1600000	Periph->ASRam[0..0x80000]	
SDRAM (16Mbyte) (optional)	0x2000000	Periph->SDRam[0..0x1000000]	CE2
SBSRAM (1Mbyte) (optional)	0x3000000	Periph->SBRam[0..0x100000]	CE3

TABLE 8. M62 External Peripheral Memory Map

This section does not describe peripheral hardware specifications and other hardware issues. Refer to the *M62 Hardware* section of this manual for additional hardware information.

Digital Input/Output

The digital input/output (I/O) buffers provide a means for generating 32 bits of direct digital input or output to and from external hardware. This I/O can be clocked from either the ‘C6201 processor or from external TTL sources, allowing external devices to automatically latch data into the I/O buffers for the ‘C6201 to read.

Input/output direction for either half of the 32-bit port may be programmed on the fly using on-board logic. The port may be configured in the software for input or output in groups of eight bits.

Memory Mapped Digital I/O Access. The following table shows the memory locations used to interact with the digital I/O buffers. Three C language routines are supplied to interact with the digital I/O port.

Function	C Language Mnemonic
Digital I/O Data Register	`Periph->Dio.Data`
Digital I/O Direction Control (4 bytes)	`Periph->Dio.Direction`
Digital I/O Latch Control	`Periph->Dio.LatchControl`

TABLE 9. Digital I/O Access Memory Location

The `Periph->Dio.Data` location is used to access the data lines of the digital I/O port. Results of read and write accesses depend on the I/O direction of the port (see below for information on setting the port direction). If the port is configured for input, a read access latches new read data from the external pins and the new data is read into the 'C6201. If the port is configured for output, the most recently latched output data is read into the 'C6201 (output data does not change). Write accesses to an input port cause no change to the port status, while write accesses to an output port cause the new data to be latched and output to the external I/O pins.

The `Periph->Dio.Direction` location controls the direction of each byte of the digital I/O port. The four least significant bits of this register are used to configure each of the bytes of the digital I/O port for either input or output, as follows:

Dio.Direction-Register Bit #	Value	Direction
0	0	DX[0..7] output (default)
	1	DX[0..7] input
1	0	DX[8..15] output (default)
	1	DX[8..15] input
2	0	DX[16..23] output (default)
	1	DX[16..23] input
3	0	DX[24..31] output (default)
	1	DX[24..31] input

TABLE 10. Table 17: Digital I/O Direction Configuration

The `Periph->Dio.LatchControl` location controls the method of latching data into each byte of the digital I/O port. The four least significant bits of this register are used to configure the latch method as either internal (triggered by CPU accesses) or external (triggered by an external TTL pulse), as follows:

Dio.LatchControl-Register Bit #	Value	Bits Affected	Clock Source
0	0	0..7	Internal (CPU-based)
	1		External
1	0	8..15	Internal (CPU-based)
	1		External
2	0	16..23	Internal (CPU-based)
	1		External
3	0	24..31	Internal (CPU-based)
	1		External

TABLE 11. Digital I/O Latch Configuration

C Language Digital I/O Functions. Data may be read or written to the digital I/O port using the following routines in the DIGITAL support library.

Function Name	Description
`DIO_dir()`	Sets the direction of all four bytes of the onboard 32-bit digital I/O port.
`DIO_read()`	Returns current state of all 32-bits of digital I/O port.
`DIO_write()`	Sets current state of all 32-bits of digital output port currently configured for output.
`DIO_latchcontrol()`	Sets the latch method of all four bytes of the onboard 32-bit digital I/O port.

TABLE 12. Digital I/O Library Functions

Timers

The timers provide the capability to generate hardware timebases, which can be used to trigger processor interrupts, analog signal conversions, or as direct outputs to external hardware. There are a total of six timebase sources built in to the M62: two 32-bit timers internal to the ‘C6201 processor, and three 16-bit channels implemented with custom logic within the FPGA plus one AD9850 direct digital synthesizer. The supplied library functions initialize the timers to a free-running, pulse generation mode suitable for generating convert pulses to the analog hardware.

The timers are initialized by code in the `timebase()` routine each time it is called. Normally, no other function calls are necessary to use the timers. However, when supplying an external TTL signal to the ‘C6201 TCLK0/1 inputs in order to provide an external timebase to analog circuitry, it will be necessary to create and use a custom version of `timebase()`, which tristates the TCLK output driver to avoid contention with external sources. Please note that certain hardware setups might be required depending on the application. See the *M62 Hardware* section of this manual for more details on how to set up the M62 board.

C Language Timer Functions. The following functions give high-level access to the timer hardware. See the appendices for complete information on the functions.

Function Name	Operation
`timebase()`	Configures a specified timer channel (0..5) l for periodic counting at a specified frequency using a specified source clock rate.

TABLE 13. C Language Timer Functions

`timebase()` can be used to set a particular timebase to a particular frequency. For example, the following call sets PIT timer channel 1 to generate a 1000 Hz output pulse stream, assuming the hardware default 1 MHz input clock to the FPGA logic:

```
timer(1, 1000.0, 1.0);
```

Memory Mapped Timer Access. It is possible to directly access to the internal timer hardware controls via memory mapped registers at specific addresses. It may be necessary to use these addresses to set the timers to a custom mode. In general, unless custom functionality is required of the timers, it is recommended that the user exclusively access the timers via the `timer()` routine rather than programming the control and period registers manually.

For information about the ‘C6201 internal timers, please see the *TMS320C6x User’s Guide*. For additional information about the custom PIT counter/timer device, contact Innovative Integration. For an example of direct timer channel control, refer to the source code for the `timebase()` function, located in the `PERIPH\DIGITAL` subdirectory.

STDIO Communication. C stdio terminal emulation is provided in the Peripheral Library. The stdio library communicates with the host TERMINAL.EXE program via the V360 PCI interface mailbox registers to provide stdio support to DSP applications running on the M62. The stdio interface may be used for real-time, non-intrusive software debugging or to create a basic user interface for OEM applications.

The following list shows the available Peripheral Library calls and their operation. See the Appendix for complete information on the functions.

Function Name	Operation
`putchar()`	Emits an 8-bit character to the terminal emulator
`getchar()`	Gets an 8-bit character from the terminal emulator’s keyboard buffer
`gets()`	Inputs a string into a target buffer
`puts()`	Displays a string from a target buffer
`sprintf()`	Formats a string into a memory buffer pointed to by buffer
`printf()`	Prints a formatted string to the terminal
`scanf()`	Inputs a formatted string from the terminal into a buffer
`sscanf()`	Converts a formatted string in memory into a buffer
`stdio_reset()`	Resets the terminal emulator display
`fopen()`	Opens a file on the Host PC, returning the file handle
`fclose()`	Closes a previously opened Host PC file.
`fread()`	Reads file contents into a target buffer
`fwrite()`	Writes a target buffer into a Host PC file

fseek()	Repositions the Host PC file pointer
ferase()	Erases the specified Host PC file
kbd_hit()	Returns a nonzero value if characters are currently available in the monitor keyboard buffer
kbd_key()	Returns 16-bit IBM scancode for pending keystroke from the terminal emulator's keyboard buffer.
gotoxy()	Moves the terminal cursor
wherexy()	Returns the terminal cursor position
clreol()	Clears to end of current line
clrscr()	Clears the terminal screen
type()	Types formatted, null terminated string to console
bold()	Enables bold text attribute in terminal emulator
normal()	Enables standard text attribute within terminal emulator
get_attribute()	Returns the current character display attributes
set_attribute()	Sets the current character display attributes
cursor()	Enables/disables the cursor
get_busmaster_addr()	Obtains the base of the host busmaster memory from the terminal emulator.
plot()	Plots a Host PC file as a graph.
view()	Plots a target buffer as a graph.

TABLE 14. STDIO Driver Functions

Using Interrupts. The M62 supports four external and numerous internal hardware interrupts. These include EI0, EI1, EI2, EI3, plus TINT0, TINT1 (internal timer/counters), internal com. port transmit and receive and DMA.

Interrupts on the TMS320C6201 may be handled by writing either high-level C or assembly language procedures within your application files, which employ the following interrupt-specific function names:

`void c_intNN()` for C handlers or

`_c_intNN` for assembly language

where NN is numbered 0 through 99 for each of the interrupts. For each interrupt, a procedure must be coded, which will be executed upon acknowledgment of interrupt NN by the 'C6201. This is described in more detail in the C Compiler Users Manual.

Consider the following code example:

```
/*

* EXAMPLE.C

*/

#define TINT0 14

main()

{
```

```
    enable_interrupts();       /* Enable unmasked xrpts */
    timebase(1, 1000.0, 1.0);  /* Internal timer 1 at 1kHz */
/* install interrupt handler on TINT1 */
    install_int_vector(c_int02, TINT0);
    enable_interrupt(TINT0);
              .
                   /* Bulk of application */
              .
       disable_interrupt(TINT0);  /* Disable TINT0 xrpt */
}
/*
*  ISR for timer 0 - Tally a variable
*/
int milliseconds
void c_int14()
{
       milliseconds++;     /* Internal timer 0 is used to */
}                          /* synthesize a timebase */
```

In this code, the internal timer 0 is configured to output a pulse every millisecond, which drives TINT0 on the ‘C6201. The vector is installed into the jump table with a call to `install_int_vector()` and the bit associated with TINT0 in the interrupt enable register, is enabled. Finally, `main()` calls `enable_interrupts()` which, sets the global interrupt enable bit so that all unmasked interrupts can be processed.

Each time the counter expires, the routine `c_int14()` executes. In this example, the variable `milliseconds` is incremented during each interrupt service cycle.

Each DSP application should include a copy of the default interrupt vector table, which is defined in `vectors.asm`. This assembly file is located in the `PERIPH\RTS` directory. When it is compiled into a `.obj` file and linked into the application, it will cause all entries in the vector table to be initial-

ized with a default handler. The one exception being the break interrupt vector, which is filled with the pointer to the talker program. If an application needs to make use of interrupts, those vectors which are affected need to be changed with `install_int_vector()` at run time.

See the target example programs provided on your distribution disks for further examples of the use of interrupts.

Example Target Programs for the M62

The following section details the example target software included with the Developer's Package. These programs are provided as models for custom user software, and it is highly recommended that the user examine these examples before beginning a first development effort for the target DSP. Full source code is provided for user inspection and reuse in modified or custom applications.

These examples will run on a standard M62 card with no additional hardware required.

HELLO

`HELLO` is a very simple introduction to basic program components and use of the C stdio library for the target card. When run with the host terminal emulator active, the program simply initializes the target hardware and stdio interface and prints the message "Hello, World" via the stdio library to the terminal emulator screen. The program then drops into an infinite dwell loop.

`HELLO` may be rebuilt from with in the Code Composer Studio environment by loading the `HELLO.MAK` project from the \target\examples directory. Then, modifying the source file `HELLO.C`, and rebuilding the project (see the Code Composer Studio documentation for more information on the application's project management and make facilities).

For correct program functionality, it is necessary to run the `HELLO` application via the host terminal emulator program. If the terminal emulator is not active and communicating with the target M62 card on which `HELLO` is running, the application will appear to hang at the first instance of a stdio function call (usually a `getint()` or `putint()` call). This is due to the fact that all stdio calls use the M62 bus mailbox interface and are handshaken with the host terminal emulator application. Any such calls will hang if the terminal emulator is not active to complete the communication link.

TEST

`TEST` is board level hardware test program, which is capable of accessing the major peripherals on the M62 to double-check proper hardware functionality. As such, it contains routines for exercising each of the peripherals on the M62, including:

1. Digital I/O
2. Internal timers
3. External timers
4. Communications Ports

Since the TEST program aims to be all-encompassing in that it tries to test as much of the board-level functionality as possible, it serves as a poor example for complicated operations such as A/D multi-channel sampling and display. However, since the code included for TEST is broken down into functional pieces, which are called separately for each subsystem to be tested, it is possible to factor out individual tests for use in other programs.

CHAPTER 8

Target DSP Peripheral Libraries

Target Functions by Category

Category	Name	Description
Board Initialization & System Functions	baud	Set baud rate on current serial port
	cpu	Set CPU number and mailbox
	cpu_num	Get CPU number
	cpu_number	Get CPU number (inline)
	detect_cpu_speed	Derive DSP clock speed
	dma_done	Wait for DMA completion
	dpram_addr	Return start address of Dualport RAM on PC31
	dpram_type	Detects 16 or 32 bit Dualport RAM on PC31
	init_serial	Initialize the serial I/O system
	InitIP	Initialize Industry Pack access structure
	mem_size	Detect size of memory space
	test_mem	PC31 memory check
Busmaster Transfer Functions	bm_init	Busmaster transfer initialization
	bm_transfer	General busmaster transfer
	fifo_init	Busmaster initialization
	transfer_complete	Wait for Busmaster transfer to complete
USB Bulk Transport Interface Functions	InitBulkTranport	Initialize the Bulk Transport Interface
	StopBulkTransport	Shut down the Bulk Transport Interface
	IsBulkTransportReady	Returns true if system can send data
	OpenBulkTransport	Opens a channel of the Bulk Transport System

	CloseBulkTransport	Shuts down an open channel of the Bulk Transport System
	ReadBulk	Read a block from a Bulk Transport channel
	WriteBulk	Writes a block to a Bulk Transport channel
	BulkDataAvailable	Returns the amount of data available for reading on a channel
	BulkSpaceAvailable	Returns the room for new data available on a channel
	FlushBulk	Forces the transmission of all data in a channel
Digital I/O Functions	C31_dig_dir	Program the direction of PC31/SBC31 PIA Digital I/O bytes
	C31_read_dig	Read PC31/SBC31 PIA Digital I/O lines
	C31_write_dig	Write to PC31/SBC31 PIA Digital I/O lines
	C31_write_dig_bit	Update a single bit on PC31/SBC31 PIA Digital I/O
	dig_dir	Program the direction of Digital I/O bytes
	read_abits	Read state of ABITS output lines
	read_abits_bit	Read state of a single ABITS output bit
	read_dig	Read Digital I/O lines
	read_dig_bit	Read state of a single digital bit
	write_abits	Write to ABITS digital output
	write_abits_bit	Update a single ABITS digital output bit
	write_dig	Write to digital output
	write_dig_bit	Update a single digital output bit
Analog I/O Control Functions	enable_analog	Initialize analog subsystem
	trigger_adc	Set triggering mode for an ADC
	trigger_adc_pair	Set triggering mode for an ADC pair
	trigger_dac	Set triggering mode for an DAC
	trigger_dac_pair	Set triggering mode for an DAC pair
	write_analog_interrupt_mask	Set which analog conversions fire interrupts
Analog Input Functions	correct_adc	Adjust ADC reading to proper range
	correct_adc_pair	Adjust a pair of ADC readings to proper range
	convert_adc	Manually trigger an ADC conversion
	convert_adc_pair	Manually trigger an ADC conversion on an ADC pair
	read_adc	Read data from ADC
	read_adc_pair	Read data from a pair of ADCs
	read_adc_automux	Read data from ADC, and switch multiplexer
	read_adc_pair_automux	Read data from a pair of ADCs, and switch mux
Analog Output Functions	correct_dac	Adjust DAC reading to proper range
	correct_dac_pair	Adjust a pair of DAC readings to proper range
	convert_dac	Manually trigger a DAC conversion

	convert_dac_pair	Manually trigger a DAC conversion on a DAC pair
	convert_dacs	Manually trigger DAC conversions using a bit mask
	read_dac	Read last value loaded into a DAC
	read_dac_pair	Read last value loaded into a DAC pair
	update_dac	Write DAC value and automatically trigger conversion
	update_dac_pair	Write DAC pair and automatically trigger conversion
	write_dac	Write value to DAC
	write_dac_pair	Write value pair to a DAC pair
Programmable Gain Functions	gain_to_mode	Convert Gain into equivalent Gain Mode number
	mode_to_gain	Convert gain mode to actual gain value
	read_gain	Read last Gain setting
	write_gain	Update gain setting for a channel
	write_gains	Update gain setting for all channels
Mux Control Functions	auto_mux	Configure automatic multiplexing feature
	read_mux	Read last setting of a particular mux
	write_mux	Update multiplexer setting for a channel
	write_muxes	Update multiplexer setting for all channels
Mailbox and Semaphore Functions	check_inbox	Check incoming mailbox for new data
	check_outbox	Check outgoing mailbox for new data
	clear_mailboxes	Clear mailboxes
	get_semaphore	Get hardware semaphore
	read_mailbox	Read from incoming mailbox
	read_mb_terminate	Read from incoming mailbox if data available
	release_semaphore	Release hardware semaphore
	write_mailbox	Write to outgoing mailbox
	write_mb_terminate	Write to outgoing mailbox if box is ready
Interrupt Support Functions	deinstall_int_vector	Remove vector from vector table
	disable_interrupt	Disable specific interrupt
	enable_interrupt	Enable specific interrupt
	host_interrupt	Target to host interrupt
	install_int_vector	Install vector into vector table
	mailbox_interrupt	Post a mailbox interrupt to the host
	mailbox_interrupt_ack	Acknowledge a mailbox interrupt
	mailbox_interrupt_deinstall	Unload the handler for mailbox interrupts
	mailbox_interrupt_disable	Disable mailbox interrupts
	mailbox_interrupt_enable	Enable mailbox interrupts
	mailbox_interrupt_install	Load a handler for mailbox interrupts
	suspend	Idle until interrupts arrive
	interrupt_cpu	Interrupt specified multiprocessor target CPU
	cpu_int_src	Return source code # for specified multiprocessor CPU

	cpu_xrpt_bit	Return register index to specified multiprocessor CPU
Timer Functions	disable_clock	Disable system millisecond timebase
	enable_clock	Initialize system millisecond timebase
	ms	Dwell milliseconds
	read_timer	Read value from a hardware timer
	timebase	Set hardware timer frequency
	timer	Set hardware timer frequency
	uclock	Get system millisecond timer value
	us	Dwell microseconds
Memory Movement Functions	copy_mem	Fast on-chip memory copy
	fill_mem	Fast on-chip memory fill
	mem_to_port	Fast on-chip transfer of data to a port
	port_to_mem	Fast on-chip transfer of data to a port
	dma_copy_mem	Fast DMA memory copy
	dma_fill_mem	Fast DMA memory fill
	dma_mem_to_port	Fast DMA transfer of data to a port
	dma_port_to_mem	Fast DMA transfer of data to a port
Conversion Functions	from_ieee	Convert from IEEE-754 floating point format
	packb	Pack byte value into int
	packh	Pack half word value into int
	to_ieee	Convert to IEEE-754 floating point format
	unpackb	Unpack byte values from int
	unpackh	Unpack half word values from int
Flash Memory Programming	fast	Restore PBCR to original value after Flash access
	flash_erase	Erase entire Flash memory
	flash_init	Initialize Flash for programming
	flash_rd	Read Flash byte
	flash_read	Read 32-bit word from Flash
	flash_sector_erase	Erase a Flash sector
	flash_wr	Write a byte to Flash memory
	flash_write	Write 32-bit word to Flash
	slow	Reduce speed of I/O accesses to access Flash memory
CPU Register I/O	clear_interrupt_flag	Disable interrupt enable bit
	get_DIE	Retrieve 320C4x DIE register
	get_IE	Retrieve 320C3x IE register
	get_IIE	Retrieve 320C4x IIE register
	get_IF	Retrieve 320C3x IF register
	get_IIF	Retrieve 320C4x IIF register
	get_IOF	Retrieve 320C3x IOF register
	get_ST	Retrieve 320C3x/4x Status register
	set_DIE	Set 320C4x DIE register
	set_IE	Set 320C3x IE register
	set_IF	Set 320C3x IF register
	set_IIE	Set 320C4x IIE register
	set_IIF	Set 320C4x IIF register
	set_IOF	Set 320C3x IOF register
	set_interrupt_flag	Set ‘C3x Interrupt Flag Bit
	set_PC	Set processor program counter
	set_ST	Set processor status register

FIFO Library Functions

FIFO Link Support	set_fifo_link_AF_levels	Set almost-full threshold levels
	fifo_link_emit	Send a character to link using handshake
	fifo_link_key	Get a character from link using hand-shake
	fifo_link_spit	Send a character to link without using handshake
	fifo_link_eat	Get a character from link without using handshake
	bleed_fifo_link	Drain FIFO into memory buffer
	fill_fifo_link	Fill FIFO from memory buffer
	reset_fifo_link	Initialize a link to empty state
	get_fifo_link_status	Obtain fullness state information
	login()	Query subordinate processors for login sequence
	sub_login	Send login sequence to master processor
	fifo_link	Return register index to FIFO link for specified CPU
FIFO Port Support	set_fifo_port_AF_levels	Set almost-full threshold levels
	fifo_port_emit	Send a character to link using handshake
	fifo_port_key	Get a character from link using hand-shake
	fifo_port_spit	Send a character to link without using handshake
	fifo_port_eat	Get a character from link without using handshake
	bleed_fifo_port	Drain FIFO into memory buffer
	fill_fifo_port	Fill FIFO from memory buffer
	reset_fifo_port	Initialize a link to empty state
	get_fifo_port_status	Obtain fullness state information

Standard I/O Library Functions

Category	Name	Description
Console Terminal Control Functions	bold	Set console text bold attribute
	clreol	Clear console to end of line
	clrscr	Clear console screen
	cursor	Enable/disable console cursor
	get_attibute	Get current console text attribute type
	gotoxy	Set cursor position
	normal	Set console text normal attribute
	set_attibute	Set current console text attribute type
	wherexy	Get cursor position
Low Level I/O	emit	Send a character to the terminal emulator
	getchar	ANSI get character from console
	kbd_hit	Install vector into vector table
	kbd_key	Get a key from the terminal emulator
	key	Get a character from the standard mail-box
	putchar	ANSI put character to console
C Standard I/O Library Emulation Functions	fclose	Close a host disk file
	ferase	Delete a host disk file by name
	fflush	Commits an open file I/O stream to disk
	fopen	Open a host disk file for read
	fread	Read from host disk file into target mem-ory
	fseek	Moves the file pointer to a specified loca-tion
	fwrite	Write to host disk file from target mem-ory
	gets	ANSI gets from console
	printf	ANSI printf to console
	puts	ANSI puts to console
	scanf	ANSI scanf from console
	sprintf	ANSI sprintf
	sscanf	ANSI sscanf
	type	Send a character string to the terminal emulator
Terminal Applet Exten-sions	get_busmaster_addr	Retrieve host busmaster address from Terminal
	plot	Transfer data buffer to host for plotting
	stdio_reset	Reset the Terminal program
	stdio_terminate	Send the termination code to Terminal

DSP Library Functions

Category	Name	Description
Signal Processing Functions	bartlett	Bartlett window generation
	bitrev	Bit reversal function
	blackman	Blackman window generation
	buffer_statistics	Calculate statistics on a data buffer
	ffft_r1	Forward Fast Fourier Transform - Real
	ffft_r2	Forward Fast Fourier Transform - Complex
	fir	Finite Impulse Response Filter
	hamming	Hamming window generation
	hanning	Hanning window generation
	harris	Harris window generation
	ifft_r1	Inverse Fast Fourier Transform - Real
	ifft_r2	Inverse Fast Fourier Transform - Complex
	vmul	Multiply two vectors into a third vector
Matrix Functions	matrix_add	Add two matrices and return a sum MATRIX
	matrix_allocate	Allocate a matrix and return its MATRIX pointer
	matrix_crop	Form sub-matrix from a larger matrix
	matrix_det	Return the determinant of a square matrix
	matrix_free	Free matrix area and MATRIX structure
	matrix_invert	Invert a square matrix, return inverse MATRIX
	matrix_mult	Multiply two matrices, return new MATRIX
	matrix_mult_pwise	Multiply two matrices element by element
	matrix_print	Print the elements of a matrix to stdout
	matrix_scale	Scale all of a matrix by a constant
	matrix_sub	Subtract two matrices and return a difference MATRIX
	matrix_transpose	Transpose a matrix, return pointer to new MATRIX
Queue Support Functions	dequeue_ptr	Remove data from a queue and adjust pointer
	enqueue_ptr	Load data into Queue and update pointers
	enqueued	Return count of data elements in a Queue
	queue_init	Initialize memory Queue structure
BERR Sequence Generation Functions	berr_decode	Tests a value in a BERR sequence
	berr_encode	Generate the next value in a BERR sequence
	berr_initialize	Set up a BERR sequence generator
Data Compression Functions	a_compress	A-Law data compression
	a_expand	A-Law data expansion
	mu_compress	Mu-Law data compression
	mu_expand	Mu-Law data expansion

CHAPTER 9

Host DLL Reference

DLL Functions Grouped by Function

The functions tabularized below may be used in any Host program written in a language, which supports access to a Dynamic Link Library. The prototypes for these functions are listed in the PERIPH\INCLUDE\LIB\TARGET.H file. The names of these functions are aliai of the actual board-specific library function names, which are proto-typed in PERIPH\LIB\HOST\ALIAS.H.

TABLE 15. Generic DLL Function List

Category	Function Prototype	Function Description
General	BOOL target_open(int target)	Opens driver for specified target DSP board. Returns boolean.
	BOOL target_close(int target)	Closes driver for specified target DSP board. Returns boolean
	LPVOID target_cardinfo(int target);	Returns address of cardinfo structure for target.
	int iicoffld(char *, int target, HWND hParent);	Loads a COFF executable file onto target DSP
Interrupt Functions	BOOL host_interrupt_enable(int target);	Enables a previously installed virtual interrupt handler.
	BOOL host_interrupt_disable(int target);	Disables a previously enabled virtual interrupt handler
	void host_interrupt_install(int target, void (*virtual_isr)(void *), void * context);	Installs a virtual interrupt handler
	void target_interrupt(int target);	Interrupts target DSP board
	void host_interrupt_deinstall(int target);	Removes a virtual interrupt handler.
	void mailbox_interrupt(int target, unsigned int value);	Interrupts the target DSP after writing value to special mailbox
	unsigned int mailbox_interrupt_ack(int target);	Acknowledges target to Host interrupt, returns special mailbox contents

Control Functions	void target_reset(int target);	Physically asserts reset on the target DSP board.
	void target_run(int target);	Deasserts reset on the target DSP board
	void target_outport(int target, int port, int value);	Outputs a value to specified DSP board I/O port address
	int target_inport(int target, int port);	Inputs a value from specified DSP board I/O port
	void target_opreg_outport(int target, int port, int value);	Outputs a value to specified DSP board operation port address
	int target_opreg_inport(int target, int port);	Inputs a value from specified DSP board operation port
	void target_control(int target, int bit, int state);	Modifies a bit in the control register of the target DSP board
Mailbox Functions	int read_mailbox(int target, int);	Reads the specified mailbox of the target DSP board
	void write_mailbox(int target, int, int);	Writes to the specified mailbox of the target DSP board.
	BOOL check_outbox(int target, int);	Interrogates the specified output mail-box status
	BOOL check_inbox(int target, int);	Interrogates the specified input mail-box status
	int read_mb_terminate(int target, int, int *, int wide);	Reads the specified input mailbox, if full
	int write_mb_terminate(int target, int box_number, int value, int wide);	Writes to the specified output mail-box, if empty
	void clear_mailboxes(int target);	Clears all mailboxes to empty state
	int target_key(int target);	Reads terminal mailbox, returns an 8-bit contents
	void target_emit(int target, int value);	Writes 8-bit value to terminal mailbox
	void target_Tx(int target, int value);	Writes 32-bit value to terminal mail-box
	int target_Rx(int target);	Reads 32-bit value from terminal mailbox
Bulk Transport Interface Functions	int BULK_GetNumDevices();	Returns the number of SBC62 USB devices detected
	BOOL BULK_OpenDevice(int iDevice, HANDLE *phDevice)	Opens a device for BULK transport access.
	BOOL BULK_CloseDevice(IN HAN-DLE hDevice)	Closes a device for BULK transport access
	BOOL BULK_OpenChannel(int iDevice, WORD wChannel, BOOL fOverlapped, BULK_HANDLE *pHandle);	Opens a data channel in BULK mode
	BOOL BULK_CloseChannel(BULK_HANDLE Handle)	Closes a data channel opened with BULK_OpenChannel()
	BOOL BULK_Read(BULK_HANDLE Handle, LPVOID lpBuffer, DWORD dwNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVER-LAPPED lpOverlapped);	Reads a block of data in BULK mode.

	BOOL BULK_Write(BULK_HANDLE Handle, LPCVOID lpBuffer, DWORD dwNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVER-LAPPED lpOverlapped);	Writes a block of data in BULK mode
	BOOL BULK_GetOverlappedReadResult(BULK_HANDLE Handle, LPOVERLAPPED lpOverlapped, LPDWORD lpNumberOf-BytesTransferred, BOOL bWait)	Gets the WIN32 Overlapped Result for the Read portion of the data channel.
	BOOL BULK_GetOverlappedWriteResult(BULK_HANDLE Handle, LPOVERLAPPED lpOverlapped, LPDWORD lpNumberOf-BytesTransferred, BOOL bWait);	Gets the WIN32 Overlapped Result for the Write portion of the data channel.
	BOOL BULK_CancelIo(BULK_HANDLE Handle)	Cancels all pending I/O on the device
	BOOL EXPORT STREAM_Open(int iDevice, WORD wChannel, WORD wBuffer-Size,WORD wBlockSize, BULK_HANDLE *pHandle)	Opens s data channel in STREAM node.
	BOOL STREAM_Close(BULK_HANDLE handle)	Closes a STREAM data channel
	WORD STREAM_WriteAvailable(BULK_HANDLE handle)	Returns the amount of space available for Write data
	WORD STREAM_ReadAvailable(BULK_HANDLE handle)	Returns the amount of data available on the STREAM channel
	WORD STREAM_Write(BULK_HANDLE handle, INT32 *pBuffer, WORD wElement-Count)	Writes a block of data to the STREAM channel
	void STREAM_Read(BULK_HANDLE handle, INT32 *pBuffer, WORD wEle-mentCount)	Reads a block of data from the STREAM channel
	void STREAM_Flush(BULK_HANDLE handle)	Writes all the output data to the target
Sempahore Functions	void get_semaphore(int target, int sema-phore);	Gains ownership of specified target semaphore
	void request_semaphore(int target, int semaphore);	Requests ownership of specified target semaphore
	BOOL own_semaphore(int target, int semaphore);	Interrogates ownership status of specified semaphore
	void release_semaphore(int target, int semaphore);	Relinquishes control of specified semaphore
Talker Functions	int target_check(int target);	Interrogates for Talker running on target
	void start_app(int target);	Starts a previously downloaded target application program
	int start_talker(int target);	Starts the target Talker executing.
	int target_revision(int target);	Returns the revision of the target Talker

CHAPTER 10

DOS Environment Requirements

Innovative Integration Developers Packages, including the TI C Compiler, make use of environment variables in order to locate header files monitor script files, etc. Be sure to set the following environment variables when installing either the C compiler or I.I. libraries. Note that several of these environment variables may be automatically set when running the SETUP program on the distribution disks. However, when upgrading from previous versions or when mixing development components from II or other sources, problems can arise.

Use the table below to insure that you specify all needed environment variables.

Environment Variable Name	**Products Affected**	**Suggested settings**
DSP_COMPILER	All TI C Compilers	**set DSP_COMPILER=**<compiler dir> ie set DSP_COMPILER=c:\c6xtools
II_BOARD	Dev Pkg Applets	**set II_BOARD=**<*board dir*> ie set II_BOARD=c:\M62cc
C_DIR	All TI C Compilers All II peripheral libraries	**set C_DIR=**%ii_board%;%ii_board%\include\target;<compiler dir> ie set C_DIR=c:\M62cc;c:\M62cc\include\target;c:\c6xtools Specified order is critical!
C_OPTIONS	TI Flt Pt C Compiler	**set C_OPTIONS=**<switches> ie set C_OPTIONS =-q –x2 –o2 –g –ss
A_DIR	All TI Assemblers	Same as C_DIR above
D_DIR	TI Debuggers (Not Code Composer)	**set D_DIR=**<*debugger dir*> ie set D_DIR=c:\c3xhll

D_SRC	All Debuggers	**set D_SRC=**<*source code dir1>;<dir2>;...;<dir n*> ie set D_SRC=c:\M62cc\stdio;c:\M62cc\dsp; c:\M62cc\periph\analog;c:\M62cc\periph\digital;… ;c:\M62cc\periph\bus
PATH	All II products All TI Tools	**set path=**<old path>;<compiler dir>;<board dir>;<host lib dir> set path=%path%;%dsp_compiler%;%ii_board%;%ii_board%\host\lib

TABLE 16. Required disk directory structure for II development tools.

The II, TI, and C Development System for the M62 requires the following environment variables be set properly for correct operation:

set dsp_compiler=c:\c6xtools

```
set ii_board=c:\M62cc

set c_dir=%ii_board;%ii_board%\include\target;c:\c6xtools

set a_dir=%ii_board;%ii_board%\include\target;c:\c6xtools

set d_src=c:\M62cc\stdio;c:\M62cc\dsp;

c:\M62cc\periph\analog;c:\M62cc\periph\digital;

c:\M62cc\periph\misc;c:\M62cc\periph\flash;

c:\M62cc\periph\bus

set c_option=-ss -o2 -g –x2 –q


path=%path%;%dsp_compiler%;%ii_board%;%ii_board%\lib\host
```

CHAPTER 11

M62/cM62 Hardware

M62/cM62 Hardware Functions

The M62 is a PCI bus compatible digital signal processor (DSP) card based around the Texas Instruments TMS320C6201 processor. Implementing a modular I/O expansion system, the M62 is particularly suited to data acquisition and control tasks, and is supported by a collection of I/O bus function cards, which provide hardware interfacing to real-world equipment.

The M62's features include:

- TMS320C6201 processor.
- Optional external zero wait-state SBSRAM and one wait-state SDRAM memory pools.
- Two inter-board communications ports (up to 80 Mbytes/sec transfer rate).
- Six channels of on-board timing (two on-chip timers, three custom 16-bit timers in FPGA logic and the 9850 DDS timebase).
- OMNIBUS module compatible (two available slots on M62, three on cM62).
- 32 bits of digital I/O.
- Two serial port connectors.
- External mux board control connectors (compatible with external TERM multiplexer and signal conditioner boards).
- JTAG hardware emulation support.

The following figure gives a block diagram of the M62/cM62.

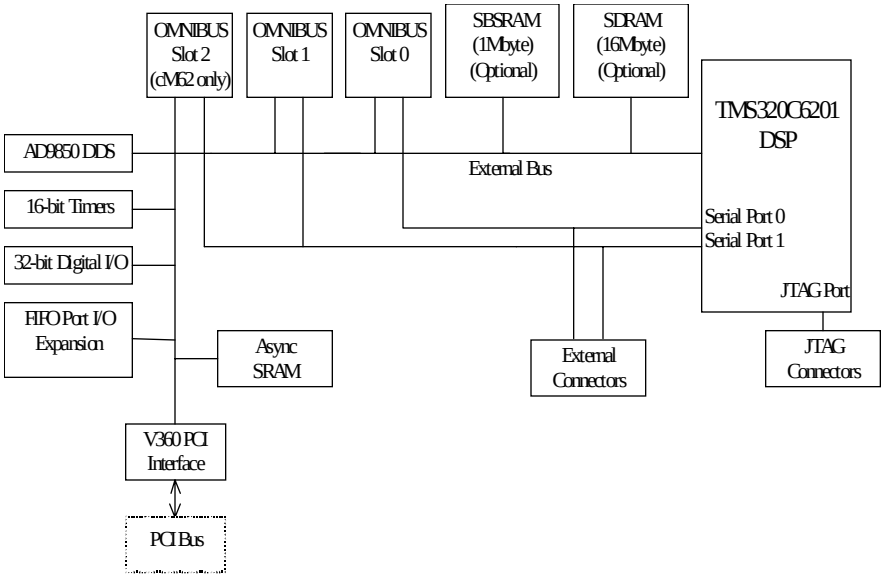


FIGURE 23. M62/cM62 Block Diagram

The cM62 is a Compact PCI compatible version of the M62 board. The cM62 retains all of the features of the M62 but is intended for use in Compact PCI host systems. In addition to the M62 feature set, the cM62 includes an additional OMNIBUS slot (allowing up to three OMNIBUS modules to be installed on a single cM62 board).

For brevity's sake, this section will refer to both cards as the M62. Any differences in functionality between the two boards (support of the third I/O bus site, different types of connectors, etc.) will be noted as required. In addition, the PCI and Compact PCI buses are collectively referred to as the PCI bus.

Memory Map

The M62 processor operates in 'C6201 HPI boot mode with memory map of type 1. In this mode, the processor's memory is available to the PCI host computer via the processor's host port interface (HPI). The on-chip memory is mapped starting at address 0. Applications programs are loaded via the HPI by the host while the card is in reset mode. Once the program is loaded, reset is deasserted by the host and the processor boots from on-chip RAM starting at address 0.

The following figure gives the processor memory map of the M62 for external peripherals and memory. Please note that this table ignores any on-chip resources.

Function	Address	Memory Space
FIFOPort	0x400000	CE0
V360 Registers	0x1400000	CE1
FIFOPort Reset	0x1410000	
FIFOPort Enable	0x1420000	
OMNIBUS Control (reserved)	0x1430000	
External Mux Control 0	0x1440000	
External Mux Control 1	0x1450000	
AD9850 Reset	0x1470000	
AD9850 Frequency Update	0x1480000	
AD9850 Write Clock	0x1490000	
Digital I/O Data Register	0x14A0000	
Digital I/O Direction Control	0x14B0000	
Digital I/O Input Latch Clock Control Register	0x14C0000	
Transmit FIFOPort PEN* Mode	0x14D0000	
Receive FIFOPort Level Status	0x14D4000	
Transmit FIFOPort Level Status	0x14D8000	
16 bit PIT Timers	0x14F0000	
External Interrupt Input 4 Select	0x1500000	
External Interrupt Input 5 Select	0x1510000	
External Interrupt Input 6 Select	0x1520000	
External Interrupt Input 7 Select	0x1530000	
OMNIBUS Strobe 0	0x1540000	
OMNIBUS Strobe 1	0x1550000	
OMNIBUS Strobe 2	0x1560000	
OMNIBUS Strobe 3	0x1570000	
OMNIBUS Strobe 4	0x1580000	
OMNIBUS Strobe 5	0x1590000	
OMNIBUS Strobe 6	0x15A0000	
OMNIBUS Strobe 7	0x15B0000	
OMNIBUS Strobe 8 (cM62 only)	0x15C0000	
OMNIBUS Strobe 9 (cM62 only)	0x15D0000	
OMNIBUS Strobe 10 (cM62 only)	0x15E0000	
OMNIBUS Strobe 11 (cM62 only)	0x15F0000	
Async SRAM (128Kx32)	0x1600000	
SDRAM (16Mbyte) (optional)	0x2000000	CE2
SBSRAM (1Mbyte) (optional)	0x3000000	CE3

TABLE 17. M62 External Memory Map

M62 Hardware Initialization Requirements

The M62 design requires the following values to be written to its hardware control registers in order to provide access to on-board hardware:

Register	Address	Value
EMIF Global Control	0x01800000	0x00003069
CE1 Control	0x01800004	0x73E70F22
CE0 Control	0x01800008	0x11010410
CE2 Control	0x01800010	0x00000030
CE3 Control	0x01800014	0x00000040
SDRAM Control	0x01800018	0x07117000
SDRAM Refresh	0x0180001C	0x00000618
Interrupt Polarity	0x019C0008	0x0000000F

TABLE 18. M62 Bus Control Register Initialization Values

These values are initialized automatically by C programs compiled under the M62 Development Package software libraries. Be sure to include initialization of these values whenever software is developed outside the Development Package or when a JTAG hardware assisted debugger is employed for code downloading to the M62. (i.e. when using Code Composer or any other JTAG debugger package)

External Memory

The M62 offers three types of external memory: asynchronous SRAM (ASRAM), synchronous DRAM (SDRAM), and synchronous burst SRAM (SBSRAM). The 128Kx32 ASRAM memory comes standard with the M62, while the SBSRAM and SDRAM are optional.

ASRAM is used by the M62 as a buffer for bus master and slave data movement on the PCI bus. The ASRAM is accessible by the V360 PCI bus interface device, allowing the processor to setup bus master data transfers, which are handled as a DMA-style transfer by the V360. The M62 utilizes the 'C6201's HOLD/HOLDA bus grant feature to provide ASRAM access to the V360. In addition, the ASRAM memory acts as a target for slave accesses by other PCI bus masters (either the host processor or other adapter cards).

The optional SBSRAM and SDRAM memories provide large, fast areas to store copious amounts of data or program information. The SBSRAM and SDRAM memories are not accessible by the PCI interface.

The 'C6201 processor operates in big endian addressing mode, allowing 8, 16, and 32 bit wide data movement to and from external SBSRAM and SDRAM memory. Async SRAM supports 32-bit accesses only, as does the V360 PCI bus interface.

M62 OMNIBUS

The M62 I/O bus provides a modular, high-speed expansion area which is directly tied to the processor's bus and which is ideally suited for I/O hardware expansion. Direct memory-mapped accesses allow the processor to transfer data to and from I/O bus peripherals constructed as plug-in modules, which can be mixed and matched to suit the particular user's functional requirements.

The OMNIBUS slots are accessed as memory-mapped peripherals with the M62 providing four decoded chip select signals per slot. The following figure gives the memory map for the OMNIBUS slots, and shows the decoded signal to slot mapping.

Function	Starting Address	Module Slot
OMNIBUS Strobe 0	0x1540000	0
OMNIBUS Strobe 1	0x1550000	0
OMNIBUS Strobe 2	0x1560000	0
OMNIBUS Strobe 3	0x1570000	0
OMNIBUS Strobe 4	0x1580000	1
OMNIBUS Strobe 5	0x1590000	1
OMNIBUS Strobe 6	0x15A0000	1
OMNIBUS Strobe 7	0x15B0000	1
OMNIBUS Strobe 8 (cM62 only)	0x15C0000	2
OMNIBUS Strobe 9 (cM62 only)	0x15D0000	2
OMNIBUS Strobe 10 (cM62 only)	0x15E0000	2
OMNIBUS Strobe 11 (cM62 only)	0x15F0000	2

TABLE 19. M62 I/O Bus Memory Mapping

Each module site provides a 32-bit wide data bus connection to the processor's data bus, with 12 bits of low-order address signals for additional decoding beyond the four chip select signals available per slot. Each module also connects to a 'C6201 serial port (serial port zero for slot zero, and serial port 1 for slots 1 and 2) to allow serial port driven I/O. Bus reset, RDY, R/W, and processor clock signals are available, as are power connections for digital 5V and analog +-5V and +-15V. Timebase connections include timer channels from both the custom 16-bit timers and the 9850 direct-digital synthesizer.

Each OMNIBUS slot has a 50 pin undedicated connector (JP17 on slot 0, JP21 on slot 1, and JP32 on slot 2) for use in providing external I/O to/from a module installed in the slot. The slot's I/O connector is in turn pinned out to a 50 pin .100" square double row header (JP18 for slot 0, JP22 for slot 1) on the M62 and to 50 pin mini SCSI style connectors on the cM62 (JP18 for slot 0, JP22 for slot 1, and JP33 for slot 2). The M62 also provides 15 pin external connectors for each slot which allow the highest numbered 15 signals on the header connectors to be pinned out external to the host computers chassis.

Connector pinouts for the module sites are provided in the appendices. Individual pin functions are noted in the tables, and in general the OMNIBUS pinout represents a direct connection to the 'C6201 local bus.

M62 OMNIBUS Memory Mapping

Since the 'C6201 processor is a byte addressable machine, which implements its address bus based on a 32-bit transfer width (i.e. the address bus starts at A2 and separate byte enable pins are supplied to control accesses to individual bytes within the 32-bit wide location denoted by the address bus), users must take care when writing software which performs OMNIBUS accesses.

The OMNIBUS specification requires 32-bit accesses and does not support byte or half-word (16-bit) accesses. No support is included in the specification for the 'C6201's byte enable pins. This means that software performing accesses must always perform 32-bit transactions with the OMNIBUS modules.

When writing C code for the M62, programmers should use only variables of type int or unsigned int (or their derived types). All accesses should be word justified (the least significant nibble of the address must always be a multiple of four). Accesses generated using pointers to variables of type char, short, or long will cause erroneous non-32-bit accesses. Correct OMNIBUS module operation under these situations is not guaranteed.

It should be noted that memory decoding within the OMNIBUS decode regions uses 32-bit addressing and that the memory map tables given in the *OMNIBUS Hardware Manual* should be treated appropriately. For example, the description of the OMNIBUS DIG module notes that the byte 3 direction control register for a module installed in site 0 is mapped to address IOMOD2 + 3. This address should be literally interpreted as 0x156000C, where IOMOD2 is equal to 0x1560000 and the offset adds decimal 12 (three 32-bit words of offset). IOMOD2 + 3 should NOT be interpreted as 0x1560003, since the offset is 3 32-bit words, not 3 bytes.

This addressing is most easily handled in C by using integer pointers and integer pointer arithmetic, which will always result in the required address alignment. For example, the following code defines a pointer and accesses the byte 3 direction control register with the documented offset:

```
unsigned int *pointer = 0x1560000;

*(pointer + 3) = 0x0;        /* set byte 3 to output mode */
```

The actual accessed memory location is 0x156000C, due to the way pointer math is handled in C.

OMNIBUS Power

The OMNIBUS interface provides five separate power supplies for use by modules along with two separate ground return connections. The following table lists the supplies and their power ratings. A separate digital 5V supply is provided along with separate digital grounds to minimize the digital noise present on the analog power supplies.

Pin Name	Voltage	Current Rating (max)
DVCC	5V (digital)	(System dependent)
+12	12V	(System dependent)
-12	-12V	(System dependent)
AVCC	5V (analog)	500 mA
-AVCC	-5V	500 mA
+AV	+15V	330 mA
-AV	-15V	330 mA

TABLE 20. I/O Bus Power Ratings

Please note that the AGND and DGND busses are separated on the M62 and for proper ground referencing they must be tied together on modules which use the analog power supplies (any supply other than digital 5V, 12V, or –12V). Innovative Integration recommends that a ferrite bead (Panasonic EXCELSA35V or equivalent) be used on custom modules to connect the two ground busses in order to prevent high frequency digital noise on the DGND bus from polluting the clean AGND return.

FIFOPort I/O Expansion

The FIFOPort feature provides a buffered bidirectional 16-bit interface which allows external hardware or other M62 boards to communicate with the M62 at high data rates. A single input FIFO is provided to buffer incoming strobed parallel data, while a FIFOPort compatible output supports clocking data to external hardware or other FIFOPorts. Access to the 'C6201 timer I/O pins is provided to support simple bit I/O requirements.

The following diagram illustrates the FIFOPort's operation. The FIFO buffer memory serves to clock incoming data and store it for use by the 'C6201. Data is formatted as a 16-bit wide data bus synchronous with an rising edge strobe signal, which acts as the FIFO load clock. The output portion consists of the same two signals: output data plus the strobe signal for the receiving end of the port.

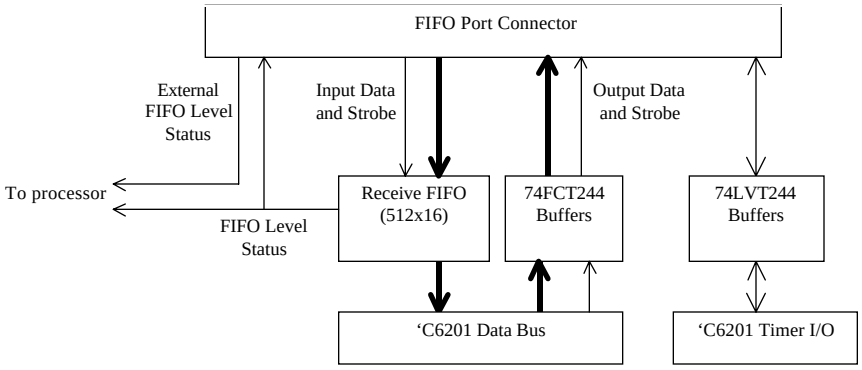


FIGURE 24. FIFOPort Block Diagram

The FIFOPort also provides external access to receive the FIFO's empty, full, and programmable almost full flags to allow hardware to monitor the FIFO's level status. The port can also receive FIFO level status from external hardware to allow the 'C6201 processor to monitor level status of FIFOs located off the M62 card. Both the onboard receive FIFO level status and the off board FIFO status lines may be polled or may generate interrupts to the 'C6201 processor.

In addition to the FIFO data management functions, access to the 'C6201 timer I/O pins is provided to support simple bit I/O requirements. The timer I/O pins are buffered through LVT family logic buffers and driven on or off the card for use where individual bit I/O control is needed for the external hardware.

Also available on the FIFOPort connector is an external interrupt input, which is connected to the processor's interrupt switch matrix. This external interrupt input allows the 'C6201 to receive an active low interrupt from external hardware.

Transmitting and Receiving FIFOPort Data

Data is transmitted and received on the FIFOPort by means of processor address location 0x400000. EMIF read and write accesses (due to either CPU or DMA activity) cause read and write strobes to be generated to the FIFOPort circuitry only when this address is accessed.

In the case of a write access, an active high output strobe is generated on the external connector and 16-bit bus data is driven out to the output bits. This data should be latched by external hardware on the rising edge of the FIFOPort output strobe. Write accesses do not affect the current state of the receive FIFO.

In the case of a read, an input read strobe is generated to the receive FIFO and its output data latched by the processor. If the data item being read in the current cycle is not the last item stored in the buffer, the next data item is clocked out by the FIFO and held ready for the next read access by the processor. Read accesses do not generate output strobes to the external connector.

Please note that the data returned by the FIFO on a read access is present on the least significant 16-bits of the processor's data bus. The most significant 16 bits are not driven and are not defined. If 32-bit CPU accesses are being used to read data from the FIFO, then the upper 16 bits of the result should be masked off before use. The DMA controller may be programmed for 16-bit access width and will automatically perform 16-bit to 32-bit data translation. Each stored 32-bit wide data item retrieved will be the concatenation of two 16-bit values read from the FIFO.

If the receive FIFO grows empty, the last data item's value will be output on any subsequent read accesses.

Monitoring FIFO Status

The FIFOPort provides a FIFO level monitoring feature, which allows software to read the receive FIFO's level indicators as well as FIFO level data from external hardware (if connected). The receive FIFO's empty, full, and programmable almost full flags can be read at any time by the CPU. The interrupt selection matrix may also be programmed to notify the CPU of level events via an interrupt (see Interrupts section for more information). The same functionality is provided for the external FIFO, allowing the CPU to read back or be interrupted by any of the six different level state conditions.

The FIFO level status is monitored using two registers, one for the receive FIFO and one for the transmit FIFO (if connected). The register bit definitions are given below.

Bit Number:	31-4	3	2	1	0
Bit Field:	Reserved	RCV_AF	RCV_FULL	RCV_HF	RCV_EMPTY

FIGURE 25. Receive FIFOPort Level Status Register

Bit Field Name	Function
RCV_EMPTY	Receive FIFO Empty Flag (1 = empty, 0 = not empty)
RCV_HF	Receive FIFO Half-full Flag (1 = not half full, 0 = at least half full)
RCV_FULL	Receive FIFO Full Flag (1 = not full, 0 = full)
RCV_AF	Receive FIFO Almost-full Flag (1 = almost-full, 0= not almost-full)

TABLE 21. Receive FIFOPort Level Status Register Definition

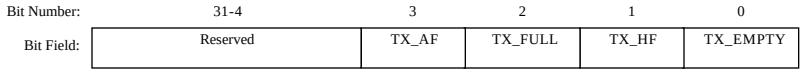


FIGURE 26. Transmit FIFOPort Level Status Register

Bit Field Name	Function
TX_EMPTY	Transmit FIFO Empty Flag (0 = empty, 1 = not empty)
TX_HF	Transmit FIFO Half-full Flag (0 = not half full, 1 = at least half full)
TX_FULL	Transmit FIFO Full Flag (1 = not full, 0 = full)
TX_AF	Transmit FIFO Almost-full Flag (1 = almost-full, 0= not almost-full)

TABLE 22. Transmit FIFOPort Level Status Register Definition

The receive FIFO level bits are read directly from the FIFO hardware on the corresponding FIFOPort, while the transmit FIFO bits are read from the level input pins on the FIFOPort connector. If no external status is being reported by the hardware connected to the FIFOPort, then these bits will read as ones (onboard 10K pullup resistors hold the transmit input pins high). If external FIFO level reporting is not desired, the level inputs may be used for application specific bit inputs to report other hardware status conditions or trigger interrupts on the M62 processor. Note that this is in addition to the dedicated timer I/O pins and the processor interrupt input pin on the FIFOPort connector, which remain available regardless of the use of the FIFO status inputs. The digital return levels given for the transmit FIFO assume connection to another ‘C6x card manufactured by Innovative Integration, or to hardware emulating similar FIFO level reporting.

With appropriate programming, the FIFO levels may also be monitored using processor interrupts. The three status bits for each FIFO in each direction are available as sources to the interrupt selection matrix for each processor. This technique is typically used to drive DMA transfers to and from the FIFOPort, where one FIFO status interrupt triggers one or more transfers using DMA synchronization. Alternatively for CPU interrupts where the target CPU in a transfer wants to be interrupted when data (or space) is available in the FIFO. This would be typical of “one-shot” FIFO transfers, where a single full FIFO’s worth of data is transferred at once. The receiving processor needs to be notified when the FIFO reached the full state so that a read operation on the other side of the FIFO may commence. For more information on using the FIFO levels to trigger interrupts to the ‘C6201 processors, see the Interrupts section.

FIFOPort Reset

The receive FIFO may be cleared and its condition reset at any time by accessing the FIFOPort reset register at address 0x1410000. The data written to the register is not critical: a write access of any data to this address will reset the FIFO. Upon reset, the FIFO levels are cleared, the flags change to reflect the FIFO empty status, and the programmable almost full control variables are reset to default values (see below for more information).

FIFOPort Enable

After a board reset or power up and prior to reading data from the receive side of the FIFOPort, software must enable data output by accessing the FIFOPort enable register at address 0x1420000. Either a read or write access to the register may be used to enable the FIFOPort. Data reads issued to the FIFOPort prior to enabling the port will clock buffered data out of the port (if any data is stored in the FIFO) but the data will not be read correctly by the processor.

Controlling the FIFOPort Programmable Almost-full Flag

In addition to the fixed function empty and full flags, the FIFOPort provides a programmable almost-full flag, which can be used to enable notification on partial FIFO transfer lengths. This feature is particularly suitable to DMA block transfers on the FIFOPort because it maximizes the transfer rates on both sides of the FIFO by keeping the buffer partially filled.

The almost-full flag operates as follows: given two initialization bytes (X and Y), the FIFO outputs an almost-full/almost-empty flag function, which is active whenever the FIFO contains X or less words of data or 512-Y or more words of data. By programming the X value equal to the almost-full level and the Y value to zero (0), the FIFO's programmable flag effectively becomes a variable partial full indicator. For example, programming the X variable to 128 and the Y variable to zero (0) yields a quarter-full output function.

The programmable almost-full flag control variables for the transmit half of the FIFOPort are initialized by enabling PEN mode, then writing the variables to the FIFOPort. PEN mode is enabled by writing a zero to the transmit FIFOPort PEN mode register at address 0x14D0000. The X variable is then written to the FIFOPort, followed by the Y variable, with both data values being 8-bits wide and right justified on the bus. The default values for the X and Y variables are both 64 (the FIFO reverts back to these values on a reset). Following the completion of the Y variable write, PEN mode should be disabled by writing a one to the PEN mode register. Please note that the almost-full flag variables may only be written immediately after a FIFO reset has been issued to the transmit side FIFO and before any data is written to the transmit FIFO.

Note: the above description of the PEN mode register operation was a change to the M62 control logic made in April 1999. Boards purchased earlier than this date should be returned to Innovative for an update. Please contact Innovative with questions concerning this feature.

Please note that this initialization operation only affects the transmit FIFO (i.e. the FIFO on the external hardware or other M62 or Quatro62 card). The FIFOPort architecture does not allow the onboard pro-

cessors to initialize the programmable levels of the FIFOPort receive FIFOs. This initialization is always performed by the external hardware prior to writing data to the receive FIFO.

Timer I/O and the FIFOPort

The FIFOPort also provides a connection to the processor's timer I/O pins. This allows designers of hardware connecting to the FIFOPort easy access to four bits of unidirectional I/O for control purposes and status reporting. The on-chip timers of the 'C6201 may be programmed to generate or receive clock and count events on the pins, or the pins may be used for general purposes bit I/O.

The M62 implements LVT family buffering between the timer I/O pins and the FIFOPort connector. Output and input levels are TTL compatible, but the outputs will not drive beyond 3.3V on the high side, and are tolerant of input voltages of up to 5V. This feature makes the FIFOPort timer I/O pins suitable for direct interfacing to 3.3V or 5V TTL compatible logic. Such logic families as HCT, LSTTL, FCT, ABT, and ACT may be directly connected to the FIFOPort timer I/O pins.

Designing External Hardware for use with the FIFOPort

Use caution when designing external hardware, which is to be connected to the FIFOPort. The signals present on the interface connector are extremely high speed and failure to handle them appropriately can cause functional problems with the FIFOPort as well as the M62's onboard components. Innovative does not recommend driving cables directly as capacitive load and ringing issues can cause corruption of the transmitted data. FIFOPort connector pinouts have been provided in the appendices.

The M62 provides mechanical mount holes suitable for use in attaching daughterboard style printed circuit boards to the FIFOPort connector. The combination of the FIFOPort connector retention and the mount hole positioning allow designers to easily create interface modules for use in adapting the FIFOPort connector to external hardware.

The following diagram gives mechanical dimensions for a FIFOPort compatible daughter PC board.

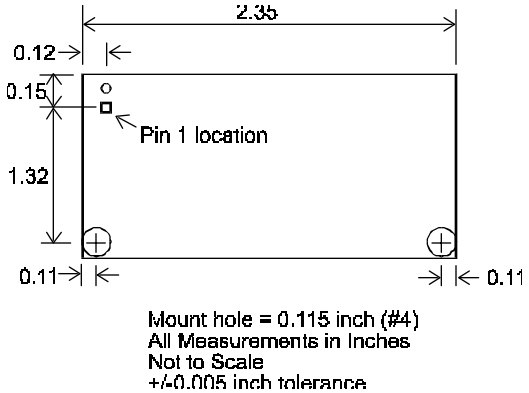


FIGURE 27. FIFOPort Daughterboard Mechanical Dimensions

FIFOPort Timing

The following diagrams give timing information for the FIFOPort circuitry. This data is derived from device specifications and is not factory tested.

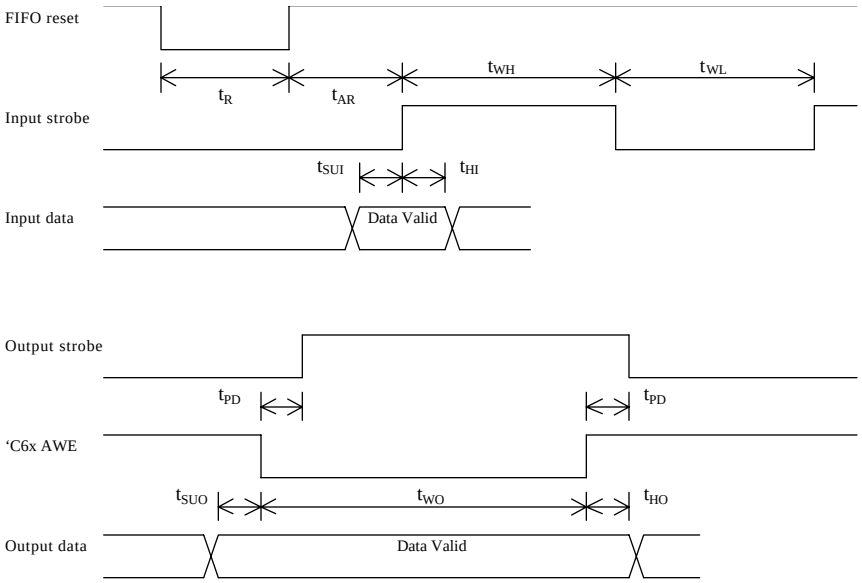


FIGURE 28. FIFOPort Timing

Parameter	min (ns)	max (ns)
t_{WH}	7	
t_{WL}	7	
t_{SUI}	5	
t_{HI}	0	
t_R	10	
t_{AR}	5	
t_{PD}		7
t_{SUO}	10[1]	
t_{HO}	0[1]	
t_{WO}	10[1]	

TABLE 23. FIFOPort Timing Parameters

Notes: Dependent on EMIF programming for CE0 space as well as processor cycle frequency. These values are determined from recommended EMIF register values.

Serial Ports

The ‘C6201’s on-chip serial ports are pinned out to connectors JP15 (port 0) and JP16 (port 1) for use with external hardware. The serial ports are also connected to the OMNIBUS slots for use with modules designed to interface to the processor serially.

Pinouts for the serial port connectors are given in the appendices. Innovative recommends buffering these ports with off board hardware in order to preserve signal integrity.

The following diagram shows the mechanical dimensions for a suggested printed circuit board outline for use in providing buffering of serial bus signals to external hardware.

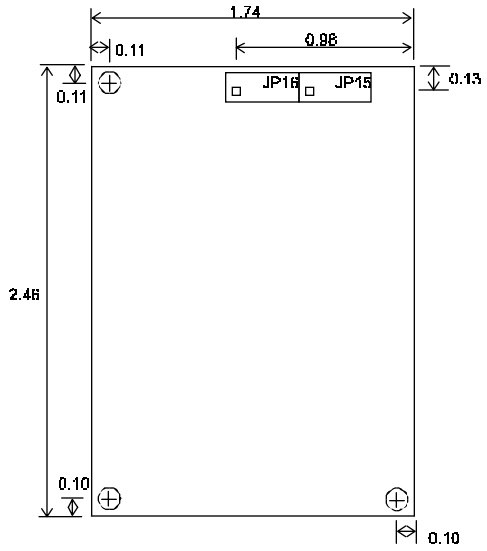


FIGURE 29. Serial Port Daughterboard Mechanical Dimensions

Timers

The M62 provides a total of six channels of independent timebase generation on board for use in timing data acquisition, servo controls, real-time counters, and other applications. The functionality is divided into three devices: two 32-bit timer channels on the ‘C6201 processor, three custom 16-bit timer channels in external logic, and a 32-bit direct digital synthesizer (DDS) channel in the AD9850 device. This section discusses the AD9850 and external timers: for more information on the on-chip timers, see the *TMS320C6201 Peripherals Reference Guide*.

On-chip Timers

The on-chip timers are available for use as software timebases and interrupt generators. They are also pinned out to connector JP31 for use with external hardware. All four processor timer pins are available on JP31, allowing applications to use each timer as a time base output, an event counter input, or as bit I/O. The timer pins are also available on the FIFOPort connector for use in controlling external hardware attached to the FIFOPort. Refer to the appendix for pinout descriptions.

16-bit Timers

The M62 implements three, custom, 16-bit timers onboard, external logic which are capable of triggering processor interrupts and acting as clock sources for the I/O modules and external hardware. The timers provide readback capability for the current count register, which allows them to be used as digital event counters. Each channel may be driven either by an onboard 10 MHz clock source or by external clock input. In addition, external gating signal are available for each timer channel which allow an external TTL signal to selectively enable or disable the timer's clock input to control counting.

All three timers consist of 16-bit decrementing free-running counters with matching 16-bit period registers driven by a source timebase. The timers decrement once per input clock until they reach zero, whereupon they automatically reload from the period register and continue counting down. The timer output is normally high and falls low for one source clock cycle upon expiration of the counter value. The source clock may be selected from either a 10 MHz onboard timebase or an external input pin. The source clock is optionally gateable via a second set of gate inputs. The gating and clock input options allow the external timers to act as event counters for external hardware.

The following table shows the memory map for the timer control registers.

Function	Address
Clock Mode	0x14F0000
Channel 0 Period	0x14F0008
Channel 1 Period	0x14F000C
Channel 2 Period	0x14F0010
Channel 0 Count	0x14F0014
Channel 1 Count	0x14F0018
Channel 2 Count	0x14F001C

TABLE 24. External Timer Control Registers

The clock mode register controls the source of the clock used to drive each channel. Data bus bit D0 controls channel zero, D1 controls channel one, and D2 controls channel two. Writing a zero to a bit selects the onboard 10 MHz clock source as the timers clock input, while writing a one selects the external INCLKx inputs available on the digital I/O connector. The INCLKx inputs allow for each of the three channels to be driven by independent clocks from external hardware. For example, writing the hex value 0x2 to the clock mode register selects the 10 MHz clock as the source timebase for channels zero and two, while channel one is driven by the INCLK1 signal.

The timer period registers are used to store the period values for each timer channel. Each timer's output pulse period is equal to the period register value plus two source clock cycles. For example, if the clock mode register for channel zero was programmed to select the 10 MHz clock as the source clock for channel zero, and the period register were programmed with the value 98. Then timer channel zero's output pulse would occur with a period of (98 + 2) source clock cycles, or a frequency of

10 MHz/(98 + 2) = 100 kHz

The highest legal value for the period register is 65534 (yielding a lowest possible output frequency of 152 Hz when using the 10 MHz onboard source clock). Please note that a period register write causes an immediate counter reload (i.e. the counter immediately starts counting down from the new period value).

The timer gate inputs allow external signals to control when the counter will decrement. Pulling the gate line low will disable clocking of the appropriate timer channel. The gate inputs are individually pulled up to 5V via a 10K resistor.

The timer output signals (TMR0, TMR1, and TMR2 for channels zero, one, and two respectively), input clock gating signals (GATE0, GATE1, and GATE2), and input clocks (INCLK0, INCLK1, and INCLK2) are all available for external connections on the digital I/O connector. Refer to the appendices for details on the pinouts.

AD9850 Direct Digital Synthesizer

The AD9850 direct digital synthesizer (DDS) is a precision programmable clock source, which is capable of generating frequencies in the range of 0 to 25 MHz with a resolution of 0.019 Hz/step. Unlike a digital counter-timer chip, which uses a digital counter to divide down a high input clock rate, the DDS uses phase-locked-loop synthesizer technology to tune a sine wave oscillator based on a 32-bit digital word. This method realizes a linear output frequency over input range rather than the nonlinear one associated with counter-timer chips, whose resolution drops dramatically as the period register used to program them falls. The counter-timer device has a nonlinear frequency step change over its input code range, as opposed to the DDS device, which maintains a linear frequency step for each input code increment. This results in the counter-timer's increased resolution at the high end of its input code range, with a correspondingly low resolution at the low end. The AD9850 timebase should be selected for use when a fairly fast but very precise and accurate clock is required by the application.

The AD9850 is mapped into I/O space as shown in the table below. The device is interfaced using the parallel I/O method, with an address to write data, one to trigger frequency/phase updates, and one to control the reset pin of the device.

Function	I/O Space Address
AD9850 Reset	0x1470000
AD9850 Frequency Update	0x1480000
AD9850 Write Clock	0x1490000

TABLE 25. AD9850 Control Registers

The write clock address latches frequency/phase data into the AD9850 one byte at a time. The least significant eight bits of the processor bus carry the bytewide data. The frequency update address causes the output frequency and phase of the DDS clock to update to the values contained in its input latches. The reset address causes an active high reset pulse to be generated to the AD9850. All three registers are write-only.

The M62 Development Package includes a routine (`timebase()`) which makes it easy to set the AD9850's output frequency.

Digital I/O

The M62 includes 32 bits of software programmable digital I/O for use in controlling digital instruments or acquiring digital inputs. The digital I/O port controls are mapped into memory space using three addresses: one to read/write the digital I/O data as a single 32-bit word, one for direction control for each byte of the port, and one for controlling the source of the clock edge used to latch input data into the digital I/O port register. The following table lists the addresses and their functions.

Function	Address
Digital I/O Data Register	0x14A0000
Digital I/O Direction Control	0x14B0000
Digital I/O Input Latch Clock Control Register	0x14C0000

TABLE 26. Digital I/O Control Registers

The direction control register provides for software control of the drive direction of the port. The least significant four bits of the register control the four bytes available on the I/O port. Bit D0 sets the direction for the least significant eight bits if the port (port bits 0-7), D1 the next least significant bits (8-15), D2 the next least significant (16-23) and D3 the most significant (24-31). Each byte is individually controllable by writing a zero (to select output) or a one (to select input) to the respective bit in the direction control register. For example, if the value 0xC were written to the direction control register, bits 0-15 would act as inputs while bits 16-31 would act as outputs. All bytes default to input mode upon board powerup or reset.

The data register allows software to directly read data from port pins programmed for input, or write data to pins programmed for output. Read operations performed from the data register on port bytes programmed for output will return the current value of the digital I/O latch (i.e. the last value written to that portion of the port). For example, suppose that the direction control was programmed to 0xC and the data register written with the data word 0x12340000. Since the most significant 16 bits are setup as outputs, those pins on the port connector would assume the value 0x1234. A subsequent read of the port would yield the value 0x1234xxxx, where xxxx is the value of the signals present on the digital I/O connector.

The input latch clock register allows the user to select either software read clocking or external hardware clocking. Writing a zero to the register selects software clocking, while writing a one selects external hardware clocking. If software clocking is selected, then the port latches programmed for input

will clock in the digital data present on the external pins at the beginning of a read cycle executed on the port data register (30-50 ns before the data is returned to the processor, depending on processor clock speed). If external clocking is selected, then the port will latch data on the falling edge of the TTL signal EXT_DIG_RD_CLK* on the digital I/O connector. The data will be held for the processor to read until the next low-going edge of the EXT_DIG_RD_CLK* signal. In the external hardware clocking mode, read operations by the processor do not affect the contents of the digital I/O latch. The latched data may be reread as many times as is required, and only another EXT_DIG_RD_CLK* pulse will cause new data to be latched into the port.

The 'FCT16952 devices used to implement the digital I/O port are capable of sourcing 32 mA and sinking 64 mA per pin.

Digital I/O Timing

The following diagram gives timing information for the digital I/O port when used in external readback clock mode (see above for details). This data is derived from device specifications and is not factory tested.

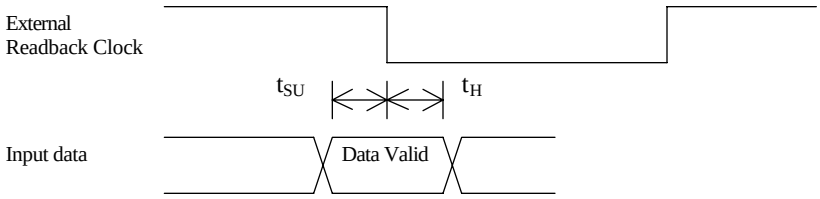


FIGURE 30. Digital I/O Port Timing

Parameter	min (ns)
t_{SU}	0
t_H	10

TABLE 27. Digital I/O Port Timing Parameters

External Mux Control

The M62 provides two external multiplexer control bus connectors for use with the TERM line of external multiplexer boards. Control for the multiplexer connectors is provided at the addresses listed in the following table.

TERM Module	Function	I/O Space Address
0	Mux #0 Channel Select	0x14D0000
	Mux #1 Channel Select	0x14D0004
	Mux #2 Channel Select	0x14D0008
	Mux #3 Channel Select	0x14D000C
	All Muxes Channel Select	0x14D0010
	Reset	0x14D001C
1	Mux #0 Channel Select	0x14E0000
	Mux #1 Channel Select	0x14E0004
	Mux #2 Channel Select	0x14E0008
	Mux #3 Channel Select	0x14E000C
	All Muxes Channel Select	0x14E0010
	Reset	0x14E001C

TABLE 28. TERM Function Memory Map

The control connectors (JP25 for TERM module 0 and JP26 for TERM module 1) select multiplexer channel numbers. The first four addresses from the start of each mux control address map allow the selection of incoming signals on each multiplexer device on the TERM. The fifth address location allows the simultaneous selection of the same channel on all multiplexer devices. The remaining address performs a global reset of the TERM hardware.

Refer to the *OMNIBUS Manual* for additional information regarding the use of Innovative's TERM modules with the M62.

Interrupts

The 'C6201 processor implements four interrupt input pins, which allow external hardware events too directly trigger software activity. Processor interrupt inputs are supported on the M62 through a set of control registers and multiplexers, which allows application software to dynamically select the source of the signal which will drive each particular interrupt input.

The available interrupt source signals are as follows:

1. External interrupt input pins 0-3 (from the I/O modules).
2. External timer channels 0-2.
3. 9850 direct digital synthesizer clock.
4. PCI bus interrupt.
5. Various receive and transmit FIFOPort level status.

The following table shows the addresses of the control registers for each processor interrupt input. A value written to the appropriate control register causes the interrupt mux to select the interrupt source given in the next two table (see below). Note that the selections vary depending on which interrupt input is being programmed.

Function	Address
External Interrupt Input 4 Select	0x1500000
External Interrupt Input 5 Select	0x1510000
External Interrupt Input 6 Select	0x1520000
External Interrupt Input 7 Select	0x1530000

TABLE 29. External Interrupt Input Control Registers

Interrupt Control Register Value	Interrupt Source
0	External Interrupt Input 0
1	External Interrupt Input 1
2	External Interrupt Input 2
3	External Interrupt Input 3
4	External Timer 0
5	External Timer 1
6	External Timer 2
7	9850 DDS Clock
8	PCI bus
9	Receive FIFOPort empty
10	Receive FIFOPort half full
11	Receive FIFOPort full
12	Receive FIFOPort almost-full
15	Deactivated (interrupt held high)

TABLE 30. Interrupt Source 4 and 5 Select Register Values

Interrupt Control Register Value	Interrupt Source
0	External Interrupt Input 0
1	External Interrupt Input 1
2	External Interrupt Input 2
3	External Interrupt Input 3
4	External Timer 0
5	External Timer 1
6	External Timer 2
7	9850 DDS Clock
8	PCI bus
9	Transmit FIFOPort empty
10	Transmit FIFOPort half full
11	Transmit FIFOPort full
12	Transmit FIFOPort almost-full
15	Deactivated (interrupt held high)

TABLE 31. Interrupt Source 6 and 7 Select Register Values

For example, if the application requires the output from external timer channel two to drive processor interrupt input five, the value six should be written to memory location 0x1510000. All interrupt control registers default to setting 15 (disabled) on powerup or board reset. Note that the processor interrupt signals generated by the logic are active low (falling edge trigger), and the 'C6201 interrupt polarity control register must be programmed to the value 0xF to correctly receive interrupts.

JTAG Test Bus

The M62 implements a JTAG 1149.1-compatible scan path loop through the onboard 'C6201, with connector compatible with the specification provided in the *TMS320C6201 User's Guide*. When connecting a JTAG controller card cable (from an Innovative Integration Code Hammer debugger card, Texas Instruments XDS-510, or other vendor's JTAG hardware), the JP11 connector is used. A shunt should always be installed on jumper JP13 when the JTAG debugger is in use.

Note: the M62 design boots the 'C6201 processor using the HPI boot mode. On device power up or reset, it is not possible to start JTAG debugger software until after the HPI boot process has been completed. The software will return with "cannot init target" error message if it is started after the processor has been released from reset, but before the processor has finished the boot process. Innovative Integration includes a small bootstrapping utility (BOOT.EXE) in the M62 Zuma Toolset which will bootload a small test application onto the M62 and which should be used prior to starting the debugger after a reset.

M62 PCI Bus Features

The M62 uses the V360 PCI bus bridge chip, along with external glue logic and asynchronous SRAM, to implement its interface to the PCI bus. The V360 acts as a bridge chip to translate accesses from the PCI bus into accesses on the 'C6201 bus. It also performs DMA style data transfers between PCI address space and the M62's asynchronous SRAM. Access to the 'C6201's HPI port through the V360 is used by host applications to bootload software into the 'C6201.

PCI Bus I/O and Memory Map

The M62 uses the V360 base address registers and address apertures to map three sets of functionality into PCI bus I/O and memory space: the V360 internal registers, the async SRAM memory, and the processor's host port interface. Address assignments are made to the board via PCI configuration cycles on system powerup, or by the host operating system. The following descriptions of the addressed features assume a working knowledge of PCI plug and play technology as well as any host operating system support provided by the system in use. The M62 Development Package provides host drivers and access support, which is highly recommended to shorten software development time.

The V360's internal register set is mapped into I/O space on the PCI bus using base address register zero (PCI_IO_BASE in the V360 data sheet), and allows host access to all of the features of the bridge chip. Host accesses to the I/O space in which the registers are mapped result in slave responses from the

V360 device. The lowest part of the register set also provides a convenient access point to the PCI configuration space registers of the device.

The async SRAM is mapped into host PCI memory using base address register one (PCI_BASE0 in the V3 literature). This allows the host processor to gain slave mode access to the SRAM memory on the M62 for data transfers, and allows other expansion boards to act as bus masters to the M62 for direct data transfers. Accesses made to the PCI mapping address by either the host processor or another bus master result in slave responses from the V360. An async SRAM access results in a HOLD/HOLDA arbitration request by the V360 device to the 'C6201. The slave access will be held not ready until the 'C6201 has dropped into hold mode and released access to the M62's processor bus. The V360 then completes the required transfer between the PCI bus and the async SRAM and releases the HOLD request t the 'C6201.

Please note that the 'C6201 must be in a state where it is capable of releasing bus ownership to the V360 for the PCI bus access to complete normally. Do not set the NOHOLD bit in the EMIF Global Control Register prior to attempting slave accesses from the PCI bus. Also note that the base address register used to map the async SRAM into PCI bus space requests 32-bit address mapping, which means that 32-bit capable host software is required to access the async SRAM memory.

The 'C6201's HPI feature is also mapped into PCI bus memory, using the V360's base address register 2 (PCI_BASE1 in the V360 data sheet). The following table gives the mapping of the various HPI registers within the PCI bus address space.

PCI Bus BAR1 Offset	Function
0x0	HPIC
0x4	HPIC
0x8	HPIA low half word
0xC	HPIA high half word
0x10	HPID low half word, with address autoincrement
0x14	HPID high half word, with address autoincrement
0x18	HPID low half word, without address autoincrement
0x1C	HPID high half word, without address autoincrement

TABLE 32. HPI Port PCI Bus Mapping

Accesses within the PCI mapped HPI interface result in slave responses from the M62. The various registers of the HPI interface are mapped as shown in the above table, and have the read/write limitations noted in the TMS320C62xx Peripherals Reference Guide. The HPI interface allows access to both the standard HPID interface (without address auto-incrementing) and to the HPID mapping which causes the current address register to be incremented automatically with each data access.

The HPI port interface uses software ready monitoring to poll the current status of the interface. Host software must poll the status of the HRDY bit in the HPIC register to determine if a current access is finished and a new access may be started.

M62 Bootstrapping

The M62 processor operates in HPI boot mode and supports direct host access to the processor's HPI port via memory mapped registers on the PCI bus. This feature allows the host to access any 'C6201 memory location and is intended for processor bootstrapping.

The 'C6201 boot process involves the following steps:

1. Toggle the processor reset active, then inactive.

2. Via the HPI interface, place a bootstrap compatible code image in the processor's internal memory starting at address zero.

3. Once the code has been placed in processor memory, write a one to the DSPINT bit in the HPIC register to wake the CPU from the reset state. The processor will then begin software execution starting at address zero in internal memory.

Although the HPI MAP1 boot mode begins running software from onchip memory, it is possible to load code anywhere in offchip memory. This is provided that the bus control registers for the memory area in question are initialized prior to any writes via the HPI interface.

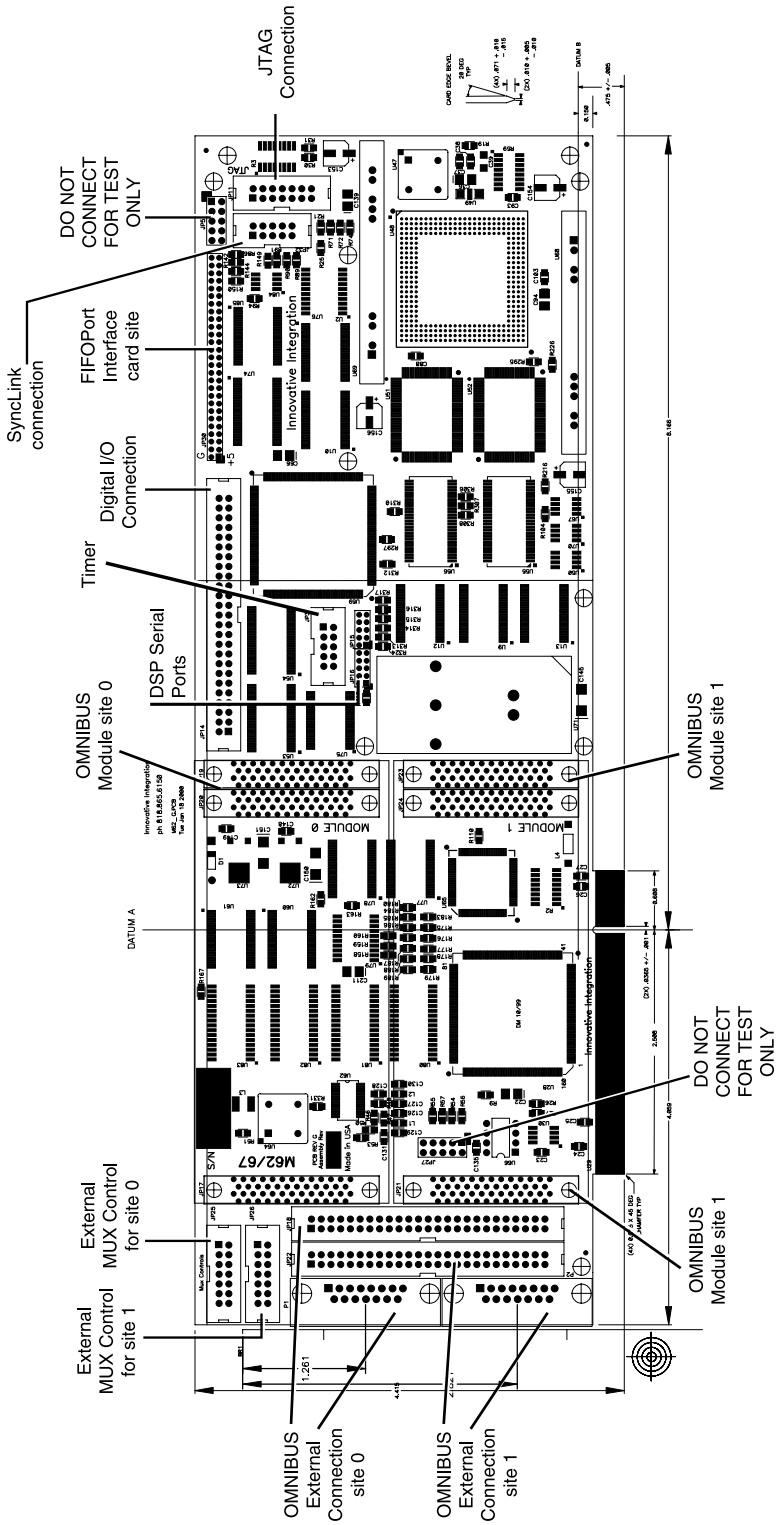
There is a complete host COFF compatible M62 bootload routine included in the M62 Development Package which facilitates 'C6201 processor bootloading.

CHAPTER 12

Appendices

Board Layout

A schematic of the board layout is displayed on the following page. Please review this schematic to familiarize yourself with the circuit board's configuration.


JTAG Connection
DO NOT CONNECT FOR TEST ONLY
SyncLink connection
FIFOPort Interface card site
Digital I/O Connection
Timer
DSP Serial Ports
OMNIBUS Module site 0
OMNIBUS Module site 1
DO NOT CONNECT FOR TEST ONLY
OMNIBUS Module site 1
External MUX Control for site 0
External MUX Control for site 1
OMNIBUS External Connection site 0
OMNIBUS External Connection site 1

Connector pinouts

JP17, JP18, JP21, JP22, P1, P2 - OMNIBUS I/O Connectors (M62 only)

Connector types: JP17, JP21: AMP .05 Subminiature D male

JP18, JP22: .100” header

P1, P2: Male DB15 connector

Number of pins: JP17, JP21: 50

JP18, JP22: 50

P1, P2: 15

Mating connector: JP17, JP21: AMP 173279-3

JP18, JP22: AMP 1-746285-0

P1, P2: AMP 747909-2

The following table shows the interconnections between the JP17 (OMNIBUS slot 0) and JP21 (OMNIBUS slot 1) module I/O connectors and their respective external I/O connectors, JP18 and P1 (OMNIBUS slot 0) and JP22 and P2 (OMNIBUS slot 1).

JP17, JP21 Pin Numbers	JP18, JP22 Pin Numbers	P1, P2 Pin Numbers
1-35	1-35	NA
36	36	1
37	37	9
38	38	2
39	39	10
40	40	3
41	41	11
42	42	4
43	43	12
44	44	5
45	45	13
46	46	6
47	47	14
48	48	7
49	49	15
50	50	8

TABLE 33. OMNIBUS I/O Connector Pinouts

JP17, JP18, JP21, JP22, JP32, JP33 - OMNIBUS I/O Connectors (cM62 only)

Connector types: JP17, JP21, JP32: AMP .05 Subminiature D male

JP18, JP22, JP33: AMP Amplimite Series III

Number of pins: JP17, JP21, JP32: 50

JP18, JP22, JP33: 50

Mating connector: JP17, JP21, JP32: AMP 173279-3

JP18, JP22, JP33: AMP 750737-5

The following table shows the interconnections between the JP17 (OMNIBUS slot 0), JP21 (OMNIBUS slot 1), and JP22 (OMNIBUS slot 2) I/O connectors and their respective external I/O connectors, JP18 (OMNIBUS slot 0), JP22 (OMNIBUS slot 1), and JP33 (OMNIBUS slot 2).

JP17, JP21, JP32 Pin Numbers	JP18, JP22, JP33 Pin Numbers
1-35	1-35
36	36
37	37
38	38
39	39
40	40
41	41
42	42
43	43
44	44
45	45
46	46
47	47
48	48
49	49
50	50

TABLE 34. OMNIBUS I/O Connector Pinouts

The following diagram gives the physical pin locations for JP18, JP22, and JP33 connectors on the cM62 board. Please note that these physical pin positions do not use the same numbering scheme as standard SCSI 50 pin connectors.

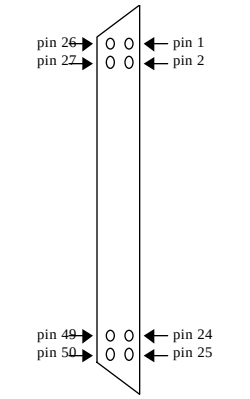


FIGURE 31. OMNIBUS I/O Connector Pin Configuration

JP19, 20, 23, 24, 34, 35 - OMNIBUS Bus Connectors

Connector types:	AMP .05 Subminiature D male
Number of pins:	50
Mating connector:	AMP 173279-3

The following table gives the pin numbers and functions for the JP19 (OMNIBUS slot 0), JP23 (OMNIBUS slot 1), and JP34 (OMNIBUS slot 2) (available only on the cM62) connectors. The functions for JP23 and JP34 are identical to those of JP19, except where noted.

Pin Number	JP19 Function	JP23 Function	JP34 Function (cM62 only)	Direction (from M62)
1, 19	Digital +5V			O, power
2, 20	Digital ground			O, power
3-18	Data bus 0-15			I/O
21, 43, 40, 45, 39, 26, 27	Address bus 2-8			O
28	Reset (active low)			O
29	External interrupt 0	External interrupt 2	External interrupt 4	I
30	Bus ready (active low)			I (open-collector)
31	Processor CLKOUT2 / 4			O
32	PIT timebase channel 0			O
33	R/W*			O
34	9850 timebase			O
35-38	IOMOD0-3 decoded selects (active low)	IOMOD4-7 decoded selects (active low)	IOMOD8-11 decoded selects (active low)	O
25	-12V			O, power
23	+12V			O, power
41,42	Analog ground			O, power
22,24	Analog -15V			O, power
44,46	Analog +15V			O, power
47,49	Analog +5V			O, power
48,50	Analog -5V			O, power

TABLE 35. I/O Module Bus Connectors

The following table gives the pin numbers and functions for the JP20 (OMNIBUS slot 0), JP24 (OMNIBUS slot 1), and JP35 (OMNIBUS slot 2) (available only on cM62) connectors. The functions for JP24 and JP35 are identical to those of JP20, except where noted.

Pin Number	JP20 Function	JP24 Function	JP35 Function (cM62 only)	Direction (from M62)
1, 3-6	Address bus 9-13			O
2, 19, 20, 49, 50	Digital ground			O, power
7-18	Reserved	Reserved	Reserved	NA
21	PIT timebase channel 1			O
22	External trigger 0	External trigger 1	External trigger 1	O
23,25	+12V (from PCI bus)			O, power
24	CLKS0	CLKS1	CLKS1	I
26	CLKR0	CLKR1	CLKR1	I/O
27	FSR0	FSR1	FSR1	I/O

28	CLKX0	CLKX1	CLKX1	I/O
29	External interrupt 1	External interrupt 3	External interrupt 5	I
30	DR0	DR1	DR1	I
31	FSX0	FSX1	FSX1	I/O
32	DX0	DX1	DX1	O
33-48	Data bus 16-31			I/O

TABLE 36. I/O Module Bus Connectors

JP14 – Digital I/O Connector

Connector type: 0.100” square double row shrouded header

Number of pins: 50

Mating connector: AMP 1-746285-0

The following table gives the pin numbers and functions for the JP14 connector.

Pin Number	JP14 Function	Direction (from M62)
1-32	Digital I/O bit 0..31	I/O
33	External Trigger 0 Input (active low)	I
34	9850 DDS Clock Output	O
35	External Trigger 1 Input (active low)	I
36, 38, 40	External Timer Ch. 0, 1, 2 Clock Outputs	O
37, 39, 41	External Timer Ch. 0, 1, 2 Gate Inputs	I
42, 44, 46	External Timer Ch. 0, 1, 2 Timebase Inputs	I
45	External Digital Readback Clock (active low)	I
47	Digital +5V	Power
49	Digital Ground	Power
43, 48, 50	Reserved	NA

TABLE 37. Digital I/O Connector

JP31 – Miscellaneous Digital I/O Connector

Connector type: 0.1” square header

Number of pins: 10

Mating connector: AMP 111811-1

The following table gives the pin numbers and functions for the JP31 connector.

Pin Number	JP31 Function	Direction (from M62)
1	External Trigger 0 Input (active low)	I
2	External Timer 0 Clock Output	O
3	On-chip Timer 1 Out	O
5	On-chip Timer 1 In	I
7	On-chip Timer 0 Out	O
9	On-chip Timer 0 In	I
4,6,8	Reserved	NA
10	Digital Ground	Power

TABLE 38. Miscellaneous Digital I/O Connector

JP15, JP16 – Processor Serial Port Connectors

Connector type: 10 pin shrouded header

Number of pins: 10

Mating connector: AMP 746285-1 (for ribbon cable termination) or Samtec SSQ style (for board-board applications)

The following table gives the pin numbers and functions for the JP15 and JP16 connectors. Pin functions of JP16 are identical to those of JP15 except where noted.

Pin Number	JP15 Function	JP16 Function	Direction (from M62)
1	CLKS0	CLKS1	I
2	FSR0	FSR1	I/O
3	CLKR0	CLKR1	I/O
4	FSX0	FSX1	I/O
5	CLKX0	CLKX1	I/O
6	Digital 3.3V		Power
7	DR0	DR0	I
8	Digital 5V		Power
9	DX0	DX0	O
10	Digital Ground		Power

TABLE 39. Processor Serial Port Connector

JP11 – JTAG Debugger Connector

Connector type: 14 pin shrouded header

Number of pins: 14

Mating connector: AMP 746285-2

The following table gives the pin numbers and functions for the JP11 connector. This connector follows the recommendations given in section 13 of the *TMS320C62xx Peripherals Reference Guide*.

Pin Number	JP11 Function	Direction (from M62)
1	TMS	I
2	TRST*	I
3	TDI	I
5	Digital +3V	Power
7	TDO	O
9,11	TCK	I
13	EMU0	I/O
14	EMU1	I/O
4, 6, 8, 10, 12	Digital ground	Power

TABLE 40. JTAG Debugger Connector

JP30 – FIFOPort Connector

Connector type: 2mm header

Number of pins: 54

Mating connector: Samtec SQW style

The following table gives the pin numbers and functions for the JP30 connector.

Pin Number	JP30 Function	Direction (from M62)
1	Digital 5V	Power
2, 44	Ground	Power
3-18	Input Data Bits 15-0	I
19	Reserved	NA
20	Input Strobe	I
21	On-chip Timer 1 Out	O
22	On-chip Timer 1 In	I
23	On-chip Timer 0 Out	O
24	On-chip Timer 0 In	I
25-40	Output Data Bits 0-15	O
41	External Interrupt Input	I
42	Output Strobe	O
43	Digital 3.3V	Power
45	Half-full Flag Out	O
46	Almost-full Flag Out	O
47	Almost-full Level Control In	I
48	Full Flag Out	O
49	Almost-full Level Control Out	O
50	Empty Flag Out	O
51	Half-full Flag In	I
52	Almost-full Flag In	I
53	Empty Flag In	I
54	Full Flag In	I

TABLE 41. FIFOPort Connector

TMS320C6201 Limitations and Errata

As of this writing, the TMS320C6201 processor has several limitations and errata that can affect the maximum clock rate at which the processor can successfully run and which may impede the proper operation of certain software applications. This section discusses limitations discovered in Innovative Intergration's testing of early M62 prototype cards, as well as errata announced by Texas Instruments regarding the current 2.0 revision silicon.

This information is being supplied to current customers and potential users of the M62 in an effort to keep you informed of the state of 'C6201 processor development and any performance limitations imposed on the M62 hardware design. Innovative Intergration will continuously update this information as new data becomes available and particularly when new silicon revisions are released by Texas Instruments to us for testing.

Processor Speed Limitations and External Memory

The current revision 2.0 silicon 'C6201 devices have bus timing issues which prohibit the use of full speed external synchronous burst SRAM (SBSRAM) and synchronous DRAM (SDRAM) devices. These limitations affect the maximum processor speed Innovative will be able to ship with current processors, dependent on the external memory configuration of the M62 hardware as ordered by the customer.

Specifically, Texas Instruments has announced that SDRAM support is limited to approximately 90MHz operation (180 MHz processor speed), while SBSRAM operation is limited to 133 MHz (133 MHz processor speed). These figures were determined through board level testing at the Texas Instruments facility.

Testing of hardware at Innovative Integration shows that SDRAM is supported at least through 80 MHz (160 MHz processor speed), while SBSRAM has been tested up to a rate of 80 MHz (160 MHz processor speed with SBSRAM in half-rate mode). Testing at a processor speed of 133 MHz with SBSRAMs running in full rate mode has shown a read/write failure rate of about 10-ppm. Texas Instruments has specified that not all SBSRAMs are reliable in their tests, so it is possible that the devices Innovative is currently using may not be up to spec. Innovative is in the process of procuring SBSRAM samples of the manufacturer and type used by Texas Instruments and will continue testing of the external memory interface to determine which devices are more reliable. In addition, Innovative will be procuring additional clock source devices, which will allow the testing of intermediate processor speeds (180 MHz, for example).

Current processors are capable of 200 MHz operation and have been tested at this rate on the M62 design. These tests involve running software strictly from on-chip memory. The external peripheral interfaces (I/O bus sites, serial ports, FIFOPorts, onboard peripherals, async SRAM, PCI interface) are unaffected by the memory interface issue as they use less aggressive bus timing.

Innovative Integration's policy on processor speed is to deliver the fastest possible speed consistent with the requested external memory configuration on the board. In situations where customers order external memory, Innovative must downgrade the processor speed to match the memory interface limitations. Current processor speeds available versus memory requirements are as follows:

External Synchronous Memory Type	Delivered Processor Speed
None	200 MHz
SDRAM	160 MHz
SBSRAM	160 MHz (SBSRAM operating in half-rate mode)

As Innovative continues testing the memory subsystems of the M62, these rates may change to improve the memory access and processor speeds.

Texas Instruments Device Errata

The current Texas Instruments device errata for the most current revision silicon TMS32C6201 devices is attached below. At this time Innovative Integration does not consider these errata to be significant to the overall operation of the card design.

TMS320C6201 SILICON ERRATA

The following is a list of problems on TMS320C6201 3.1 silicon or any lower revision. TI creates a new document revision when a new silicon bug is discovered. However, TI does NOT update previously edited files. For example, if you have silicon revision 2.0 and the latest silicon revision is 3.1, you should look at the latest silicon errata for 3.1, as it will also contain any problems found in silicon version 2.0.

Silicon revision is identified by a code in the lower left-hand corner of the chip. The code is of the format Cxx-yyww. If xx is 31 then the silicon is revision 3.1. If xx is 20, 21, 30 then the silicon is revision 2.0, 2.1, OR 3.0 respectively.

The Revision ID of the CPU (which is NOT the same as the silicon revision) can be found in the Revision ID field of the Control Status Register (CSR). Please refer to the *TMS320C62x/C67x CPU and Instruction Set Reference Guide* for details about the Control Status Register. The following table shows the silicon revision and its CPU Revision ID:

Silicon Revision	CPU Revision ID found in CSR
C6201 Revision 2.1	1
C6201 Revision 3.0	2
C6201 Revision 3.1	2

The CPU Revision ID only shows the revision of the CPU. Please note that 6201 Silicon Revision 3.0 and 3.1 have the same CPU Revision ID, since the same CPU is used in both silicon versions. Users should only refer to the silicon revision number, and not the CPU Revision ID, when using this document.

Please also request the latest *TMS320C6201 Peripheral Reference Guide* and any Errata.

Note:

- New items in this document are
  - Problem 3.1.8
  - Problem 3.1.9
- Problems in revision 3.0 silicon not fixed in revision 3.1 have been re-numbered as 3.1.x problems. This creates gaps in the 3.0.x problem numbering sequence.
- All remaining 3.0.x problems are fixed on revision 3.1.
- All remaining 3.1.x problems will be fixed on a future device. At present there is no specific plan for a future version.

List Of Bugs

REVISION 3.1 SILICON BUGS

Problem 3.1.1 DMA: Issues when pausing at a block boundary

The following problems exist when a DMA channel is paused at a block boundary:

- DMA doesn't flush internal FIFO when a channel is paused across block boundary. As a result, data from old and new blocks of that channel are in FIFO simultaneously. This prevents other channels from using the FIFO for high performance until that channel is restarted. Note that data is not lost when that channel is started again. Internal reference number C601299.
- For DMA transfers with auto-initialization, if a channel is paused just as the last transfer in a block completes (just as the transfer counter reaches zero), none of the register reloads take place (count, source address, and destination address). When the same channel is restarted, the channel will not transfer anything due to the zero transfer count. This problem only occurs at block boundaries. Internal reference number C601258.

WORKAROUND: Do not pause across block boundary if the internal FIFO is to be used by other channels for high performance. For DMA transfers with auto-initialization, if a channel is paused with a zero transfer count, manually reload all registers before restarting the channel.

Problem 3.1.2 DMA: Transfer incomplete when pausing a Frame Synchronized transfer in mid-frame

If a frame-synchronized transfer is paused in mid-frame and then restarted again, a DMA channel does not continue the transfer. Instead, the channel waits for synchronization. If the channel is manually synchronized, it will properly complete the frame, but will immediately begin the transfer of the next frame too. This behavior occurs for both a software pause (setting START = 10b) and for an emulation halt (with EMOD = 1). Internal reference number C601257.

WORKAROUND:

- If pausing the DMA channel in software, do the following to restart:
  1. Set the RSYNC bit in the Secondary Control Register.
  2. Read the Transfer Count Register and then write back to Transfer Count Register. This would enable the present frame to transferred but will wait for the next sync event to trigger the next frame transfer.
  3. Set START to 01b or 11b.
- If pausing the DMA channel with an emulation halt, do the following to restart:
  1. Double-click on the Transfer Count Register and hit enter (rewrite current transfer count).
  2. Set the RSYNC STAT bit in the Secondary Control Register (change 0xXXXX4XXX to 0xXXXX1XXX).
  3. Run.

***Note that the order of 1 & 2 is critical for an emulator halt (EMOD = 1), but not for the software pause.

Problem 3.1.3 DMA Multi-frame Split-Mode transfers source address indexing not functional

If a DMA channel is configured to do a multi-frame split-mode transfer with SRC_DIR = Index (11b), the source address is always modified using the Element Index, even during the last element transfer of a frame. The transfer of the last element in a frame should index the source address using the Frame Index instead of the Element Index. DST_DIR = 11b functions properly. Internal reference number C601256.

WORKAROUND: For multi-frame transfers, use two DMA channels instead of using the split-mode. Source Index works properly for non-split-mode transfers.

Problem 3.1.4 DMA: Stopped transfer reprogrammed doesn't wait for sync

If any non-synchronized transfer (ex: Auto-init Transfer) is stopped, and then the same channel is programmed to do a Write Synchronized Transfer (ex: Split-mode transfer), the write transfer does not wait for the Sync event. Internal reference number C601261.

WORKAROUND: Perform a non-synchronized dummy transfer of one element to/from the same location before starting the synchronized transfer.

Problem 3.1.5 DMA freezes if post-increment/decrement across port boundary

For any DMA transfers with source/dest address post-increment/decrement, if the last element to be transferred is aligned on a port boundary, then the DMA may freeze before transferring this element. A port boundary is the address boundary between external memory and program memory, between external memory and the peripheral address space, or between program memory and the peripheral address space.

The following conditions cause DMA to freeze:

- For non-sync and frame-sync transfers: if a channel is paused after the second-to-last element is read, when the channel is then restarted with a request to the address at a port boundary the DMA will freeze.
- For split-mode transfers or read/write-sync transfers: the DMA will freeze while transferring the element aligned on the port boundary. A continuous burst transfer with post-increment/decrement source/dest address does not exhibit this problem. Internal reference number C601300.

WORKAROUND: Do not transfer to boundary addresses if the DMA source/dest address is post-incremented/decremented.

Problem 3.1.6 DMA paused during emulation halt

When running an auto-initialized transfer, the DMA write state machine is halted during an emulation halt regardless of the value of EMOD in the DMA Channel Primary Control Register. The read state machine functions properly in this case. The problem exists only at block boundaries. If EMOD=1, this problem is irrelevant since the DMA channel is expected to pause during an emulation halt. Internal reference number C601301.

WORKAROUND: There is no workaround for EMOD=0. Expect DMA transfers to pause when the emulator stops the processor.

Problem 3.1.7 DMA: RSYNC=10000b (DSPINT) doesn't wait for sync

If RSYNC in the DMA Channel Primary Control Register is set to Host-port host to DSP interrupt (DSPINT – 10000b), the DMA channel would do the read transfer without waiting for the sync event. There is not a problem if WSYNC is set to DSPINT. Internal reference number C601302.

WORKAROUND: Do not synchronized DMA reads to DSPINT. If a DMA read is desired during a Host-port host to DSP interrupt, set RSYNC in the Primary Control Register to one of the EXT_INT events instead (EXT_INT4 – EXT_INT7) and have the host trigger an interrupt on that pin rather then by writing to HPIC.

Problem 3.1.8 EMIF: Invalid SDRAM access to last 1kByte of CE3

If 16 Mbytes of SDRAM (2 64 Mbit in a 1Mx16x4 organization) is used in CE3 then you can have invalid accesses to the last 1kByte of CE3 (0x03FFFC00).

This occurs when the following is true:

- After a DCAB (Deactivate all pages) to all SDRAM CE spaces (forced by Refresh or MRS command)
- The first access to CE3 is to the last page of CE3 (0x03FFFC00).

Then a page activate will not be issued to CE3. Since the SDRAM in CE3 is in a deactivated state at that point, invalid accesses will occur. Internal reference number C630280.

WORKAROUND:

Best Case: Avoid designing a board with a 64Mbit (1Mx16x4) SDRAM mapped into CE3.

Alternative: If a 64 Mbit SDRAM is located in CE3, avoid using the last 1kByte in the CE3 memory map (0x03FFFC00).

Problem 3.1.9 Cache During Emulation with Extremely Slow External Memory

If a program requests fetch packet “A” followed immediately by fetch packet “B”, and all of the following four conditions are true:

1. A and B are separated by a multiple of 64k in memory (i.e. they will occupy the same cache frame)
2. B is currently located in cache
3. You are using the emulator to single-step through the branch from A to B
4. The code is running off of an extremely slow external memory that transfers one 32-bit word every 8000+ CPU clock cycles (CPU running at 200 MHz)

Then A will be registered as a “miss” and B will be registered as a “hit”. B will not be reloaded into cache, and A will be executed twice. This condition is extremely rare because B has to be in cache memory, and must be the next fetch packet requested after A (which is not in cache memory). In addition, this problem only occurs if you single-step through the branch from A to B using the emulator, AND if the code is located in an extremely slow external memory. Internal reference number C630283.

WORKAROUND:

- Do not single-step through the branch from A to B if the above conditions are true.
- Do not use an extremely slow external memory (transfers one 32-bit word every 8000+ CPU clock cycles) if conditions 1, 2, and 3 are true.

REVISION 3.0 SILICON BUGS

Problem 3.0.8 EMIF: Inverted SDCLK and SSCLK at Speeds above 175 MHz

There is a speedpath in the device that causes SDCLK and SSCLK to startup 180 degrees out of phase (effectively inverted) from the desired waveform. Normally, EMIF outputs are delayed 1/2 CPU clock from the rising edge of SDCLK/SSCLK to give it adequate hold time while maintaining more than adequate setup times.

The desired relationship is described in the TMS320C6201B datasheet (SPRS051D, p 37 and p 40) and illustrated in Figure A and Figure C below. However, in the case where SDCLK/SSCLK becomes inverted (Figure B and Figure D), control signals only have 1/2 CPU clock of setup to the next SDCLK/SSCLK rising rather than 3/2 CPU clock of setup. This has two negative affects to interface timing to external synchronous RAMs.

1) On writes, setup time to RAMs for control signals and write data is reduced by 1 CPU cycle.

Figure A. Write Example - Desired Behavior

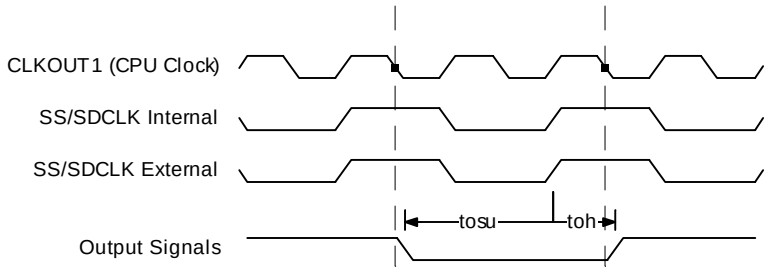
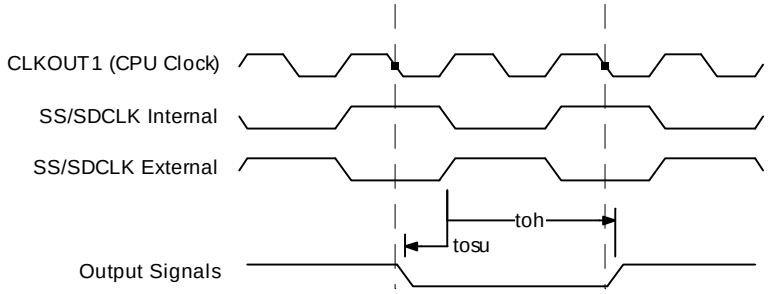


Figure B. Write Example - Failing Behavior



2) On SBSRAM/SDRAM reads, data will be sampled on the falling edge BEFORE the rising edge that would be expected. In this case, the input setup time for data at the C6x is reduced by 1 CPU cycle. Note that this case can be compounded with Case 1). The control signals could be latched one SSCLK/SDCLK cycle (2 CPU cycles) late by the memories. Thus, the setup could be reduced by up to 3 CPU cycles and be more than an entire SSCLK/SDCLK late.

Figure C. Read Example - Desired Behavior

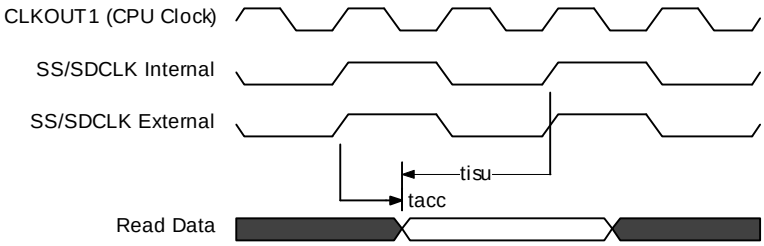
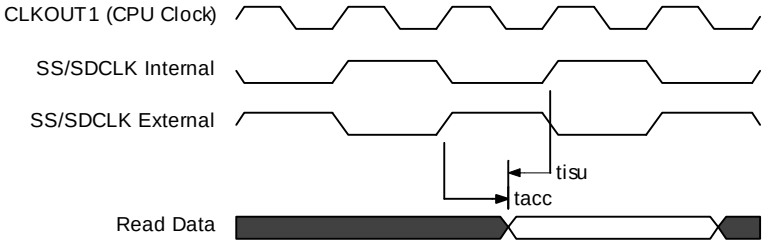


Figure D. Read Example - Failing Behavior



Note that CLKOUT2 is also affected by this speedpath bug and is 180 degree out of phase. It behaves in the same way as SDCLK. Internal reference number C601307.

WORKAROUND:

- For prototypes raising the core supply to 1.9-2.1V corrects this problem. We DO NOT recommend this in boards shipped to customers, since the manufacturing process is not designed to be reliable outside the normal operating range. This option allows the user to verify current board designs at all valid frequency ranges.
- Reduce the operating frequency of the TMS320C6201B until SSCLK/SDCLK has the desired relationship. Typically this occurs at 175 MHz across the range of recommended operating conditions.
- Since SSCLK and SDCLK are inverted externally relative to each other by design, these signals can be swapped on external memory interfaces to correct the problem (SSCLK to SDRAM and SDCLK to SBSRAM). This will cause invalid operation at frequencies below 175 MHz and will not work with future silicon revisions.
- If CLKOUT2 is used as an SDRAM clock, follow all the workarounds for SDCLK.

ALTERNATE WORKAROUNDS: The following alternate work arounds can help for certain board and layout configurations.

- Using faster (125 Mhz or PC100) SDRAMs and/or SBSRAMs will reduce the chances of data corruption and/or increase the frequency at which reliable memory operation can be observed. Operation is not guaranteed to be reliable across operating conditions and different samples of memory and 6201B devices due to lot to lot variation on both the memory and the 'C6201B.
- SDCLK/SSCLK can be delayed externally. This can be accomplished either via inverter(s), precision delay device, or longer board route on the clock line. The idea is to force the external clock to resemble the desired clock waveform as closely as possible, providing more setup for both reads and writes.
- You may start the device at a frequency where the skew does not occur and raise the operating frequency to the desired rate. This must be done at each processor reset. This solution works since the speed path exists in the reset (non-run time) operation of the SDCLK/SSCLK circuit. Whatever operations starts at reset is observable until the next reset.

RESOLUTION

- Version 3.1 of silicon will correct this problem.

Problem 3.0.9 CPU: L2-unit long instructions corrupted during interrupt

If an interrupt occurs causing a B-side L-unit (.L2 unit) instruction that writes a long value to be annulled, the top 8 bits of the result will be written rather than being annulled. This bug only applies to the B-side L-unit (.L2 unit). A-side L-unit (.L1 unit) functions correctly. Internal reference number C620774

This bug will not affect:

- Customers programming in C with no long data types.
- Customers not using code with long instructions on the .L2-unit.
- Customers only using long instructions on the .L2-unit inside loops 5 or less than 5 cycles long. (Interrupts are disabled in the 5 delay slots of a branch)

WORKAROUND:

- Disable interrupts using the appropriate compiler switches, or cregister modifications, in the affected C code.
- Disable interrupts 7 execute packets before any long instructions on the .L2-unit that are NOT in the delay slots of a branch.
- Use the .L1-unit for long instructions if interrupts are anticipated.

RESOLUTION

- Version 3.1 of silicon will correct this problem.

REVISION 2.1 SILICON BUGS

Problem 2.1.1 EMIF: CE Space Crossing on Continuous Request Not Allowed

Any continuous request of the EMIF cannot cross CE address space boundaries. This condition can result in bad data read, or writing to the wrong CE. Internal Reference Numbers 2600 & 3421.

WORK-AROUNDS:

CPU Program Fetch: The simplest fix is for all external program to reside within a single CE space. Alternatively, program fetch flow should not occur across CE spaces. This can be accomplished by branching on chip in between executing from one CE to another CE.

DMA: All DMA block transfers without read or write synchronization should have all EMIF addresses within a frame to belong to one CE space. In other words, all read (src) addresses should belong to one CE space and should not cross CE boundaries. The same applies to write (dst) addresses within a frame. Note that the source can be in the same CE space or different CE space as the destination. DMA transfers with read and/or write synchronization together with CE boundaries crossed between frames are not affected by this bug.

CPU Data Access: External CPU data accesses cannot perform continuous requests and thus are not affected by this bug.

Problem 2.1.2 EMIF: SDRAM invalid access

An invalid SDRAM access occurs when each of the following is true:

- Two or more SDRAM devices in different CE spaces
- Each SDRAM device has a page activate
- One active page is in bank 0 and the other in bank 1
- Each CE space with SDRAM is accessed (alternating) without a page miss or refresh occurring (no Deactivate command).

OR

- Two or more SDRAM devices in different CE spaces
- A trickle refresh deactivates both devices
- Before refresh occurs, a request to access one CE space comes in. The refresh will wait until the first requestor has completed.
- If request to second CE space occurs before refresh occurs, then an invalid access takes place, since the controller neglects the fact that this space was deactivated.

Internal Reference Numbers 4139, 0335, and 0871.

WORKAROUND: Avoid use of multiple CE spaces of SDRAM within a single refresh period.

Problem 2.1.4 DMA: RSYNC cleared late for Frame Sync'd transfer

In a frame-synchronized transfer, RSYNC is only cleared after the beginning of last write transfer. It should occur after the start of the first read transfer in the synchronized frame. Internal reference number 0267.

WORKAROUND: Wait until end-of-frame (perhaps using DMAC pins for external status) to issue next frame synchronization.

Problem 2.1.5 McBSP: DXR to XSR copy not generated

If any element size other than 32 bits is written to the DXR of either serial port, then the register is not copied to the XSR. Internal reference number 0511.

The following work around is **applicable only for non-split mode DMA** transfers.

WORKAROUND:

(1) For little-endian mode:

Always write 32 bits to the DXR. When using the DMA, it is possible to perform word transfers, but increment or decrement the address by one or two bytes using one of the global index registers. If the serial port is transferring out 16-bit words, which are stored on consecutive half-word boundaries in memory (either internal or external), the DMA would need to be set up such that it performs word writes to DXR (ESIZE = 00b). The global index register used would need an element index of 0x0002 (2 bytes). If an 8-bit data transfer is desired, then element index would need to be 0x0001.

Please note that this workaround assumes that the receive justification, RJUST in the McBSP's SPCR is set for right justification (zero-fill or sign-extended). If left justification is chosen for receive data, the DMA receive src address pointing to DRR should be changed to DRR+3 (which is 0x018C0003 for McBSP0 and 0x01900003 for McBSP1) for byte-size elements and DRR+2 for half-word elements. This ensures packing data on byte or half-word boundaries for receive data.

Example:
Configure the DMA as follows:
(a) For half-word / byte-size accesses with right justification on receive data:

ch_A: /* for transmit */
src_address = mem_out; dst_address = DXR;
Element_size = WORD
Address_inc_mode = index
Index_reg_value = 2 /* change this to 1 for byte writes */

ch_B : /* for receive */
src_address = DRR; dst_address = mem_in;

```
Element_size = HALF /* change this to BYTE for 8-b element size */
Address_inc_mode =  inc_by_ element_size
/* inc_by_index whose value is as specified for ch_A above will also work */
```

(b) For half-word / byte-size accesses with left justification on receive data:
Same as (1)(a) above EXCEPT for:
ch_B : /* for receive */
src_address = DRR+3; /* for byte accesses */ OR
= DRR+2; /* for half-word accesses */

(2) For big-endian mode:

Always write 32 bits to the DXR.
(a) For half-word accesses with right justification on receive data:
ch_A: /* for transmit */
src_address = mem_out;
dst_address = DXR+2; /* 0x018C0006 for McBSP0 or 0x01900006 for McBSP1 */
Element_size = WORD
Address_inc_mode = index
Index_reg_value = 2

ch_B : /* for receive */
src_address = DRR+2 /* 0x018C0002 for McBSP0 or 0x01900002 for McBSP1 */
dst_address = mem_in;
```
Element_size = HALF;
Address_inc_mode = =  inc_by_ element_size
/* inc_by_index whose value is as specified for ch_A above will also work */
```

(b) For half-word writes with left justification on receive data:
Same as (2)(a) above EXCEPT for:
ch_B : /* for receive */
src_address = DRR;

(c) For byte-size writes with right justification on receive data:
ch_A: /* for transmit */
src_address = mem_out;
dst_address = DXR+3; /* 0x018C0007 for McBSP0 or 0x01900007 for McBSP1 */
Element_size = WORD
Address_inc_mode = index
Index_reg_value = 1

ch_B : /* for receive */
src_address = DRR+3 /* 0x018C0003 for McBSP0 or 0x01900003 for McBSP1 */
dst_address = mem_in;
Element_size = BYTE;
```
Address_inc_mode = =  inc_by_ element_size
/* inc_by_index whose value is as specified for ch_A above will also work */
```

(d) For byte-size writes with left justification on receive data:
Same as (2)(c) above EXCEPT for:
ch_B : /* for receive */
src_address = DRR;

Problem 2.1.6 DMA Split-Mode End-of-frame Indexing

If a DMA channel is configured to do a multi-frame split-mode transfer, both the Receive and Transmit transfers will generate an end-of-frame condition. This will cause the FRAME COND bit to be set multiple times per frame in the Secondary Control Register of the channel.

Also, if DST_DIR = Index (11b), the end-of-frame condition by both the Receive and Transmit Transfers will cause a destination address to be incremented using Frame Index, rather than Element Index. The problem is that BOTH the last element in a frame for the Receive Read Transfer (split source to destination) AND the last element in a frame for the Transmit Write Transfer (source to split destination) will cause the destination address to be indexed using the frame index. This should only occur for the last element in a frame for the Receive Read Transfer. Internal reference number 0559.

WORKAROUND: If the FRAME COND bit is used to generate an interrupt to the CPU and/or the frame index and the element index on the destination address are not the same for a split-mode transfer, use two DMA channels.

Problem 2.1.7 DMA Channel 0 Multi-frame Split-Mode Incompletion

If DMA Channel 0 is configured to perform a multi-frame split-mode transfer, it is possible for the last element of the last frame of the Receive Read to not be transferred. After the last element of the last frame of the Transmit Write Transfer, the element count is reloaded into the Channel 0 Transfer Counter Register, which may allow for the Transmit Read Transfer to be initiated. If the read synchronization and write synchronization are far enough apart in CPU cycles, then it is possible for the DMA to hang (due to the Transmit Read) before the Receive Write gets its sync event and completes the transmission. Internal reference number 0558.

WORKAROUND: If a multi-frame split-mode transfer is required, use DMA channel 1, 2, or 3.

Problem 2.1.8 Timer clock output not driven for external clock

When FUNC = 1 (TOUT is a timer pin), if CLKSRC = 0 (external clock source) the TOUT pin is not driven with TSTAT. The timer still functions correctly, but the output is not seen externally. Internal reference number 0568.

WORKAROUND: None. Timer functions correctly.

Problem 2.1.9 Power Down pin PD not set high for Power Down 2 mode

The power down pin, PD, only goes high (active) in power down mode 3, not in power down mode 2. Internal reference number 0537.

WORKAROUND: None. Power down modes function correctly.

Problem 2.1.10 EMIF: RBTR8 bit not functional

If RBTR8=1, a requester with continuous requests will not relinquish control of the EMIF even to a higher priority requester. Internal reference number 0432.

WORKAROUND: Leave RBTR8 set to the default of 0.

Problem 2.1.11 McBSP: Incorrect μLaw companding value

The C6201 McBSP u-Law/A-Law companding hardware produces an incorrectly expanded u-Law value. McBSP receives u-Law value 0111 1111, representing a mid-scale analog value. Expanded 16-bit data is 1000 0000 0000 0000, representing a most negative value. Expected value is 0000 0000 0000 0000. McBSP expands u-Law 1111 1111 (also mid-scale value) correctly. u-Law works correctly for all encoded values, except for 0x7f. Internal Reference Number 0651.

Problem 2.1.12 Cache: False cache hit – Extremely rare

If a program requests fetch packet “A” followed immediately by fetch packet “B”, and the following are true:

- A and B are separated by a multiple of 64k in memory (i.e. they will occupy the same cache frame)
- B is currently located in cache

Then A will be registered as a “miss” and B will be registered as a “hit”. B will not be reloaded into cache, and A will be executed twice. This condition is extremely rare because B has to be in cache memory, and must be the next fetch packet requested after A (which is not in cache memory). Internal Reference Number 4372.

WORKAROUND: The program should be re-linked to force A and B to not be a multiple of 64k apart.

Problem 2.1.13 EMIF: HOLD feature improvement on revision 3

This is documented as a difference between the 320C6201 revision 2.x (and earlier) and revision 3.0 (and later).

The HOLD feature of the ‘C6201 currently will not respond to a HOLD request if the NOHOLD bit is set at the time of the HOLD request, but is then cleared while the HOLD request is pending. In other words, for a HOLD request to be recognized, a high to low transition must occur on the HOLD input while the NOHOLD bit is not set. Future revisions of the device will operate as described below.

If NOHOLD is set and a HOLD request comes in, the C6x will ignore the HOLD request. If while the HOLD request is still asserted the NOHOLD bit is then de-asserted, the HOLD will be acknowledged as expected. Internal reference number 0101.

WORKAROUND: In order to recognize a pending HOLD request when the state of the NOHOLD bit is changed from 1 to 0, a pulse must be generated on the input HOLD line. This can be done by logically OR-ing a normally low general purpose output (DMAC can be used) with the HOLD request signal from the requestor, and creating a high pulse on the general purpose output pin.

Problem 2.1.14 EMIF: HOLD request causes problems with SDRAM Refresh

If the HOLD interface is used in a system with SDRAM, there are some situations that are likely to occur.

If the NOHOLD bit is not set and an external requestor attempts to gain control of the bus via the HOLD signal of the EMIF at the exact same time as the EMIF is issuing a SDRAM Refresh command, the HOLD request is never recognized. Even if the NOHOLD bit is set in the EMIF Global Control Register, SDRAM Refreshes are still disabled as long as the HOLD request is pending. A single Refresh after receiving the HOLD request is issued, but no additional Refreshes are issued until the HOLD request is removed. The C6x still owns the bus since the NOHOLD bit is set.

In addition, if an SDRAM burst is started just prior to a HOLD request, it is possible that the request will not be recognized until a refresh occurs. This will potentially allow for the HOLD request to be ignored for several micro-seconds. Internal reference number 0757 and 0777.

WORKAROUND: Do not allow a requestor to activate the HOLD line without acknowledging it for longer than the SDRAM refresh period. A workaround can be accomplished by keeping the NOHOLD bit set and software poll the HOLD bit of the EMIF Global Control Register. Software polling of the HOLD bit in the EMIF Global Control Register will indicate when a HOLD request has been received (this can be done in the SD_INT service routine or Timer interrupt service routine).

Upon detecting a HOLD request, SDRAM refreshes are disabled, NOHOLD bit is cleared, and a pulse is generated on the input HOLD signal (can use DMACx as a general purpose output pin in combination with the requestors HOLD signal). Then NOHOLD can be set and SDRAM refreshes enabled in anticipation of the next HOLD request.

Problem 2.1.15 DMA Priority Bit Ignored by PBUS

The CPU always has priority over the DMA when accessing peripherals. The DMA PRI bit is ignored and treated as “0”. Internal reference number 0540.

WORKAROUND: Leave sufficient gaps in CPU accesses to the PBUS to allow the DMA time to gain adequate access.

Problem 2.1.16 DMA Split-Mode Receive Transfer Incomplete After Pause

If the DMA is performing a split-mode transfer and the channel is paused after all Transmit Reads in a frame are completed but before the Receive Reads are completed, then the Receive Transfer will not complete after the channel is restarted. Internal reference number 0606.

WORKAROUND: Do not pause a split-mode transfer at the end of a frame unless the frame has completed.

Problem 2.1.17 DMA Multi-Frame Transfer Data Lost During Stop

If the DMA is stopped while performing an unsynchronized, multi-frame transfer, all of the read data may not be written. The data will be written when the channel is restarted. This case will only occur when the frame size (element count) is 10 or less and data elements from multiple frames are in the FIFO when it is stopped. Internal reference number 0789.

WORKAROUND: Keep frame size > 10, synchronize the frame (FS = 1), or do not stop the transfer.

Problem 2.1.18 Bootload: HPI boot feature improvement on revision 3

This is documented as a difference between the TMX320C6201 revision 2.x (and earlier) and revision 3.0 (and later).

Currently during HPI boot, all accesses to program memory are treated as writes by the PMEMC. This means that the host may not read the internal program memory space, as doing so will overwrite the memory space, usually with all zeros. The PMEMC will be changed to differentiate between reads and writes to program memory during boot. Internal reference number 0604.

Problem 2.1.19 PMEMC: Branch from external to internal

The program flow is corrupted after branching from external memory to internal program memory when the following are true:

- CPU is executing from external memory
- A CPU stall occurs that holds the CPU until all pending program fetches complete. CPU stalls may be caused by:
  - External data access
  - Multi-cycle NOPs
  - Prolonged data memory bank conflict with DMA
  - Multiple accesses to on-chip peripherals (not likely to cause this problem)
- A branch to internal program memory is taken before a new fetch packet is requested (i.e. during the same fetch packet that is executed when the CPU stalls.

The CPU will branch correctly to the internal memory location and correctly execute the code located there. When the branch is executed to return to external memory, the CPU will not complete the branch properly and the program will crash. Internal reference number 0958

WORKAROUND: There are several workaround options, depending on the situation that causes the failure. One or more of the following should be used to circumvent the problem:

- If the problem arises during an interrupt, move IST to external memory (same CE as code).
- If the problem occurs after a branch, delay the branch instruction with single-cycle NOPs or extend the delay slots to span multiple fetch packets (i.e. follow the branch instruction with parallel NOPs).
- If an external data access is causing the CPU stall, place data in internal data memory.
- If a multi-cycle NOP is causing the stall, change to multiple single-cycle NOPs.
- If stall is due to the CPU being starved, change the DMA priority to be lower than that of the CPU.

Problem 2.1.21 DMA: DMA data block corrupted after start with zero transfer count

If DMA is stopped after it has been started with a zero transfer count, then reprogrammed and started again, the first element of the block will be corrupted. Internal reference number 0242.

WORKAROUND: Make sure the transfer count is not near zero when starting the DMA.

REVISION 2.0 SILICON BUGS

Problem 2.0.1 Program Fetch: Cache Modes Not Functional

WORK-AROUND: Use internal program memory in mapped mode.

Problem 2.0.2 Bootload: Boot from 16-bit and 32-bit Asynchronous ROMs Not Functional

16-bit wide ROM mode and 32-bit wide asynchronous mode work in run time without bugs. The problem is only in boot. . Internal Reference Number 3088.

WORK-AROUND: Place all code in the lowest byte of the boot ROM.

Problem 2.0.3 DMA Channel 0 Split Mode Combined with Auto-initialization Performs Improper Re-Initialization

The source address (transmit read address) is reset too early when both split mode and auto-initialization are enabled. The bug exists on DMA channel 0 only. Internal Reference Number 3481.

WORK-AROUND: Substitute one of the other channels for channel 0 when this configuration is desired.

Problem 2.0.4 DMA/Program Fetch: Cannot DMA into Program Memory when Running Program From External

Performing a DMA transfer into program memory while running from off-chip can cause invalid program data to read by the CPU. Internal Reference Number 2978.

WORK-AROUND: DMA into program memory only when running from internal program memory.

Problem 2.0.5 Data Access: Parallel Read and Write Accesses to Same EMIF or Internal Peripheral Bus Location Sequenced Wrong

This bug occurs under the following conditions:

- A load and store are in the same execute packet. And Either
  - The addresses both point to off-chip memory through the EMIF, and the load has a destination register in side A (thus the store would have a source register in side B). Or
  - The addresses both point to the peripheral bus, and the load has a destination register in side B (thus the store would have a source register in side A).

When these conditions occur, the store occurs first rather than the load. In general, this will only cause an error if both the load and store addresses are the same. This bug DOES NOT occur if both accesses are to internal data memory. Internal Reference Number 3087.

WORK-AROUND: Avoid loading and storing the same address on the same cycle.

Problem 2.0.7 EMIF: Reserved Fields Have Incorrect Values

Fields in Bits 15:14 of EMIF CE Space control registers are writeable. They should be read only and have a 0 value. Bits 5:4 of EMIF SDRAM control register are 11b rather than 0. Internal Reference Number s 3248, 3283.

WORK AROUND: Mask these values if 0's are expected and to only write 0's to reserved fields.

Problem 2.0.8 EMIF: SDRAM Refresh/DCAB Not Performed Prior to HOLD Request Being Granted

SDRAM is left in the current state when an external HOLD is granted. SDRAM refresh/DCAB is necessary if an interface to a shared memory external SDRAM controller is desired. Internal Reference Number 3249.

WORK-AROUND: Make sure the external controller performs a refresh/DCAB before performing SDRAM accesses.

Problem 2.0.9 McBSP New Block Interrupt does not occur for Start of Block 0

When end-of-block interrupt is selected ((R/X)INTM=01b), does not occur at end of frame (i.e. before block 0). Internal reference number 4357.

WORK-AROUND: This interrupt is used when on-the-fly channel selection/enabling is being performed. A static channel selection/enabling avoids this.

Problem 2.0.11 DMA/Internal Data Memory: First load data corrupted when DMA in high priority

In the case of a single load from A side or B side followed by two loads in parallel from both sides, and in concert with a DMA high priority access to the same bank as the parallel load, the DMEMC provides corrupt data for that first load. Internal Reference Number 3858.

```
Example:    LDW   .D1    *A3, A4 ; A4 gets corrupt data due to the bug
            LDW   .D2    *B3, B4
          || LDW   .D1    *A6, A7
```

WORK-AROUND: Avoid high priority DMA transfers to/from internal data memory during these conditions.

Problem 2.0.12 McBSP: FRST Improved in 2.1 over 2.0

The following enhancements were made in 2.1.

When /FRST transitions to a 1, the first frame sync is generated after 8 CLKG clocks. The 2.0 implementation was such that the first frame sync was generated after FPER+1 number of CLKG clocks.

/FRST=1 is valid only when /GRST=1. In other words the user has to set /FRST=1 only after /GRST=1. If not, write to /FRST=1 is ignored or rather a zero is forced on /FRST by the logic.

During normal operation, when /FRST=1 and /GRST=1, and now the user puts the sample rate generator in reset (/GRST=0) without first clearing the /FRST bit to zero, then the logic will force a zero to /FRST bit before shutting down the sample rate generator.

Problem 2.0.13 McBSP: /XEMPTY stays low when DXR Written Late

/XEMPTY goes low and stays low when DXR was written on either the last bit or next to last bit of the previous word being transferred to DX. Internal Reference Number 3383.

Problem 2.0.14 EMIF: Multiple SDRAM CE Spaces: Invalid access after refresh

This bug exists only in those systems that have SDRAMs in more than one CE space. When there are two SDRAM accesses performed to two CE spaces, followed by a refresh, the pages in all CE spaces with SDRAM are de-activated. The first CE space to be accessed after the refresh gets activated correctly. The bug is that if the second CE space is accessed on the same page as before the refresh, it will not get activated before the read or write is attempted. Internal Reference Number 3952.

WORKAROUND: Avoid use of multiple CE spaces of SDRAM within a single refresh period.

Problem 2.0.18 DMA/Internal Data Memory: conflict data corruption

This bug occurs when the CPU has high priority and is accessing a bank with word access (load or store) followed by similar (load or store) halfword access, and the DMA is also accessing the same bank simultaneously with word accesses:

```
Example:    LDW   .D1    *A3, A4
            LDH   .D2    *A3, A5; A DMA to the bank containing never completes
                                  ; but the DMA continues as if it did
```

The data transfer done by the DMA is corrupted in halfwords (or rather not updated) when the DMA transfer is complete. Internal Reference Number 4195.

WORKAROUND: When DMAing to/from internal memory with DMA in low priority, use half-word or byte element size transfers. Alternatively, avoid the above code sequence during DMA transfers.

Problem 2.0.19 EMIF: Data Setup Times

The data setup time for the external memory interface is listed in the February 21, 1998 Advanced Information TMSX320C6201 Data Sheet as 2 ns, 3ns, and 2ns for Full Rate SBSRAM, ½ Rate SBSRAM, and SDRAM respectively. In revision 2.0 of silicon, these values are to 4.8, 6.0, and 6.4ns respectively, from worst-case simulation data (low voltage, high temperature, worst case process conditions.)

WORKAROUND: In room temperature operation we have not seen these setup times affect operation except in the case of SDRAM where it may be limited to 80-95 MHz.

Problem 2.0.24 EMIF Extremely Rare Cases Cause an Improper Refresh Cycle to Occur.

If a trickle refresh is waiting for the EMIF, and the refresh timer counts down and makes the refresh urgent JUST AS the EMIF grants the request, then CE is held low for only 1/2 SDCLK cycle during the deactivate command before the refresh. This will result in an invalid deactivate command. Since the SDRAM did not deactivate the open page, the next activate command following the refresh will not be executed by the SDRAM. This will cause any subsequent accesses to go to the non-deactivated page. This will cause corrupt data read and writes if the page to be opened after the refresh was not the same page that was open before the refresh. Internal Reference Number 3453.

WORKAROUND: Increase the refresh period.

TMS320C6701 SILICON ERRATA

The following is a list of problems on TMS320C6701 silicon. TI creates a new document revision when a new silicon bug is discovered. However, TI does NOT update previously edited files. For example, if you have silicon revision 0.0 and the latest silicon revision is 1.0, you should look at the latest silicon errata for 1.0, as it will also contain any problems found in silicon version 0.0.

Silicon revision is identified by a code in the lower left-hand corner of the chip. The code is of the format Cxx-yyww. If xx=10, the silicon is revision 1. If no code is found, or if xx=00, the silicon is revision 0.

The Revision ID of the CPU (which is NOT the same as the silicon revision) can be found in the Revision ID field of the Control Status Register (CSR). Please refer to the *TMS320C62x/C67x CPU and Instruction Set Reference Guide* for details about the Control Status Register. The following table shows the silicon revision and its CPU Revision ID:

Silicon Revision	**CPU Revision ID found in CSR**
C6701 Revision 0.0	1
C6701 Revision 1.0	2

The CPU Revision ID only shows the revision of the CPU. Users should only refer to the silicon revision number, and not the CPU Revision ID, when using this document.

Please also request the latest TMS320C6000 Peripherals Reference Guide and any Errata.

Note:

- New items in this document is
  - Changes to the TMS320C6701 datasheet (SPRS067C)
  - Problem 0.0.10 description is modified
- Problems in revision 0.0 silicon not fixed in revision 1.0 have been re-numbered as 1.0.x problems:
  - Problem 0.0.16 is re-numbered as 1.0.1.
  - Problem 0.0.17 is re-numbered as 1.0.2.
- All remaining 0.0.x problems are fixed on revision 1.0.

CHANGES TO THE TMS320C6701 DATA SHEET (SPRS067C)

JTAG TEST-PORT TIMING (p. 71)

NO.	PARAMETER		'C6201, 'C6201B		UNIT
			MIN	MAX	
1	Tc(TCK)	Cycle time, TCK	50		ns
4	Th(TCKH-TDIV)	Hold time, TDI/TMS/TRST valid after TCK high	9		ns

Figure 16. SBSRAM Read Timing (1/2 Rate SSCLK)*

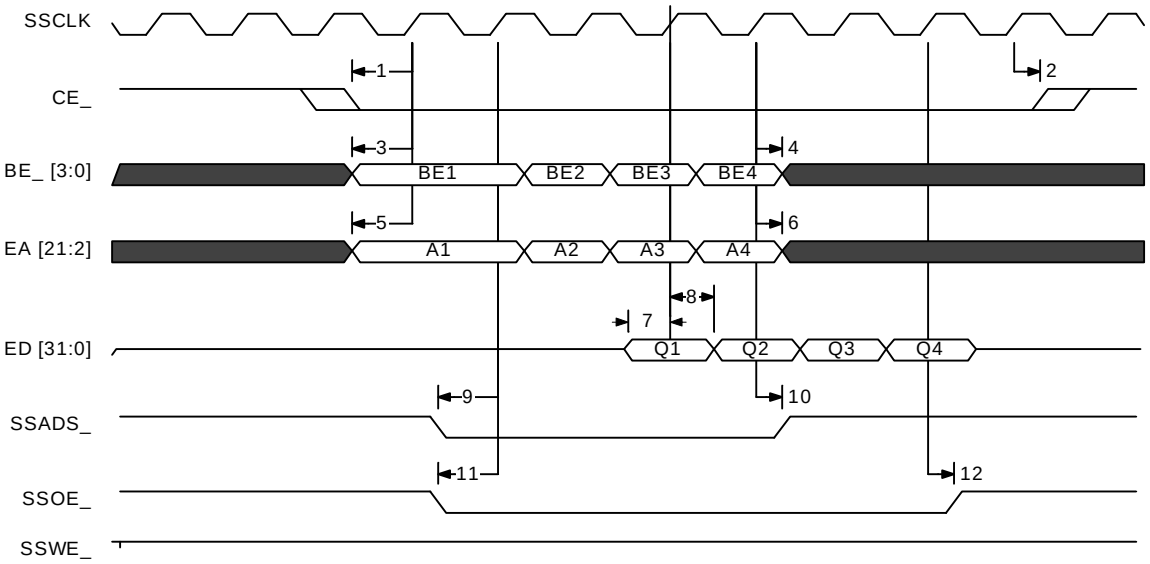
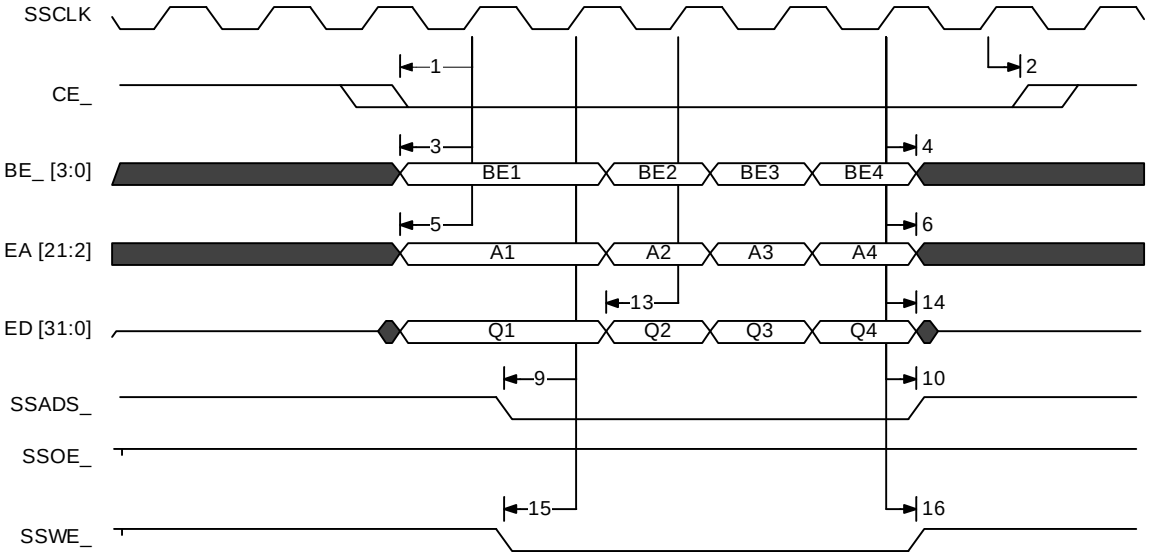


Figure 17. SBSRAM Write Timing (1/2 Rate SSCLK)*



* The /CEx output setup and hold times are guaranteed to be accurate relative to the clock cycle to which they are referenced, since these timings are specified as minimums. However, the CE output setup and hold time may be greater than that shown in the datasheet in multiples of P ns. In other words, for output setup time, the /CEx transition from high to low may happen P, 2P,…, or nP ns before the time specified by the datasheet. Similarly, for output hold time, the /CEx low to high transition may happen P, 2P, …, or nP ns after the time specified by the datasheet. This is indicated by the period of uncertainty for specs 1 and 2 in Figure 16 and Figure 17 above.

List Of Bugs

REVISION 1 SILICON BUGS

Problem 1.0.1 EMIF: Invalid SDRAM access to last 1kByte of CE3

If 16 Mbytes of SDRAM (2 64 Mbit in a 1Mx16x4 organization) is used in CE3 then you can have invalid accesses to the last 1kByte of CE3 (0x03FFFC00).

This occurs when the following is true:

- After a DCAB (Deactivate all pages) to all SDRAM CE spaces (forced by Refresh or MRS command)
- The first access to CE3 is to the last page of CE3 (0x03FFFC00).

Then a page activate will not be issued to CE3. Since the SDRAM in CE3 is in a deactivated state at that point, invalid accesses will occur. Internal reference number C630280.

WORKAROUND:

Best Case: Avoid designing a board with a 64Mbit (1Mx16x4) SDRAM mapped into CE3.

Alternative: If a 64 Mbit SDRAM is located in CE3, avoid using the last 1kByte in the CE3 memory map (0x03FFFC00).

Problem 1.0.2 Cache During Emulation with Extremely Slow External Memory

If a program requests fetch packet “A” followed immediately by fetch packet “B”, and all of the following four conditions are true:

1. A and B are separated by a multiple of 64k in memory (i.e. they will occupy the same cache frame)
2. B is currently located in cache
3. You are using the emulator to single-step through the branch from A to B
4. The code is running off of an extremely slow external memory that transfers one 32-bit word every 8000+ CPU clock cycles (CPU running at 200 MHz)

Then A will be registered as a “miss” and B will be registered as a “hit”. B will not be reloaded into cache, and A will be executed twice. This condition is extremely rare because B has to be in cache memory, and must be the next fetch packet requested after A (which is not in cache memory). In addition, this problem only occurs if you single-step through the branch from A to B using the emulator, AND if the code is located in an extremely slow external memory. Internal reference number C630283.

WORKAROUND:

- Do not single-step through the branch from A to B if the above conditions are true.
- Do not use an extremely slow external memory (transfers one 32-bit word every 8000+ CPU clock cycles) if conditions 1, 2, and 3 are true.

Problem 1.0.3 DMA: Split-Mode transfers corrupted if channel 1, 2, 3 are stopped

There is a problem with stopping DMA channel 1, 2, or 3 when operating in split-mode transfers. If the DMA split-mode receive and transmit transfers are not in sync with one another when the channel is stopped, and then the same DMA channel is programmed for a new split-mode transfer, the new transfer will execute correctly but may not terminate completely. This problem does not exist in channel 0. Internal reference number C621764.

WORKAROUND: Do not stop DMA channels 1, 2, and 3 when they are operating in split-mode.

Or manually force the number of elements received and transmitted transfers to be equal. Split-mode is most commonly used with the on-chip McBSPs. In typical McBSP applications, the transmit data is two elements ahead of the receive data. Therefore to stop the serial transfer do the following:

- Reset the McBSP to prevent additional sync events
- Set RSYNC_STAT twice for the DMA channel to force two receive transfers
- Stop the DMA channel

In order to ensure that the same number of elements are transferred, the source and destination addresses can be checked.

REVISION 0 SILICON BUGS

Problem 0.0.1 DATA MEMORY CONTROLLER: LDDW Bug

LDDW from external data memory (any CE space) fetches only the lower 32 bits instead of 64 bits. However, LDDW from internal data memory works correctly and fetches the full 64-bit data, except for any one of the following cases listed below, in which LDDW from internal data memory incorrectly fetches only the lower 32 bits instead of 64 bits:

(1) Code sequence causes

Two successive execution packets with either one of the following patterns can cause the LDDW error:

Packet	A-Side Instruction	B-Side Instruction	Comments
1	LDDW	Any store instruction	Internal data memory bank conflict
2	Any load instruction	Any store instruction	Internal data memory bank conflict

OR

Packet	A-Side Instruction	B-Side Instruction	Comments
1	Any load/store	LDDW	Internal data memory bank conflict
2	Any load/store	Any load	Internal data memory bank conflict

Both of the above code sequences cause the LDDW instruction to return corrupted data.

(2) Step mode causes

Stepping through any code sequence that contains an LDDW instruction will cause the internal LDDW error.

(3) DMA causes

If a DMA access causes an internal data memory bank conflict with another load or store instruction in the same execute packet with an LDDW instruction, the LDDW instruction will return only the lower 32 bits of data. This problem only occurs if DMA has priority, since the bug is caused by the CPU stalling. If the CPU has priority, the CPU will not stall (unless you also have cause 1 or cause 2 happening). Internal reference number 1, 3.

WORKAROUND: Do not use LDDW to fetch data from external memory. When using the compiler, allocate all accessed data to internal data memory since there is no guarantee that the compiler will not use the LDDW instruction. In addition, some of the Double Precision math library functions in rts6701.lib and rts6701e.lib are found to use the LDDW instruction. In those cases try to use the equivalent single-precision library function. For example, use "float logf(float x)" instead of "double log(double x)". When using hand-coded or linear assembly code, if it is not possible to allocate data to internal data memory, avoid using the LDDW instruction to access this data. LDB, LDH, and LDW can all be used instead.

In order to use LDDW from internal memory without failure, the user must ensure that the code pattern outlined in (1) above is never generated (note that the data bank conflicts are required in this pattern for a failure to occur).

Users may single-step code to debug, but DO NOT single step over the execution of an LDDW instruction and all 5 of the cycles of latency of the LDDW instruction. Use a breakpoint after the 5-cycle latency to resume single stepping of the program.

To use DMA in programs that use LDDW from internal memory, the user must ensure that the execute packets that contain a LDDW instruction do NOT contain another load or store access, so that DMA accesses will not cause internal data memory bank conflicts.

Problem 0.0.2 Multi-cycle stalls during internal data memory bank conflicts

Program flow will get corrupted data if ALL of the following are true:

- The program contains an execute packet with a B-side internal data load.
- This B-side internal data load is followed by an execute packet with a parallel load that generates an internal data bank conflict (address bits 1, 2, 3, and 15 are the same between the loads).
- AND a multi-cycle stall occurs during the execute of the parallel load packet.

The data for the first B-side load will be corrupted by the data for the second B-side load. The original B-side load data will be lost.

Note in this description B-side refers to the destination register for the load, NOT the D-unit or address register. Internal reference number 2.

WORKAROUND:

(1) Ensure the code does not contain any internal data bank conflicts (a brute-force method is to ensure there are no execute packets with parallel loads)

(2) Ensure the code that includes parallel loads with internal data bank conflicts will not have any stalls generated (due to external data fetches, external instruction fetches, high priority DMA activity, user single-steps or breakpoints, or any other cause of a stall). In that case, only a single-cycle stall will occur due to data bank conflicts. The program will work correctly.

Problem 0.0.3 DMA: Transfer incomplete when pausing a Frame Synchronized transfer in mid-frame

If a frame-synchronized transfer is paused in mid-frame and then restarted again, a DMA channel does not continue the transfer. Instead, the channel waits for synchronization. If the channel is manually synchronized, it will properly complete the frame, but will immediately begin the transfer of the next frame too. This behavior occurs for both a software pause (setting START = 10b) and for an emulation halt (with EMOD = 1). Internal reference number C601257.

WORKAROUND:

- If pausing the DMA channel in software, do the following to restart:
  1. Set the RSYNC bit in the Secondary Control Register.
  2. Read the Transfer Count Register and then write back to Transfer Count Register. This would enable the present frame to transferred but will wait for the next sync event to trigger the next frame transfer.
  3. Set START to 01b or 11b.
- If pausing the DMA channel with an emulation halt, do the following to restart:
  1. Double-click on the Transfer Count Register and hit enter (rewrite current transfer count).
  2. Set the RSYNC STAT bit in the Secondary Control Register (change 0xXXXX4XXX to 0xXXXX1XXX).
  3. Run.

***Note that the order of 1 & 2 is critical for an emulator halt (EMOD = 1), but not for the software pause.

Problem 0.0.4 DMA Multi-frame Split-Mode transfers source address indexing not functional

If a DMA channel is configured to do a multi-frame split-mode transfer with SRC_DIR = Index (11b), the source address is always modified using the Element Index, even during the last element transfer of a frame. The transfer of the last element in a frame should index the source address using the Frame Index instead of the Element Index. DST_DIR = 11b functions properly. Internal reference number C601256.

WORKAROUND: For multi-frame transfers, use two DMA channels instead of using the split-mode. Source Index works properly for non-split-mode transfers.

Problem 0.0.5 DMA: Issues when pausing at a block boundary

The following problems exist when a DMA channel is paused at a block boundary:

- DMA doesn't flush internal FIFO when a channel is paused across block boundary. As a result, data from old and new blocks of that channel are in FIFO simultaneously. This prevents other channels from using the FIFO for high performance until that channel is restarted. Note that data is not lost when that channel is started again. Internal reference number C601299.
- For DMA transfers with auto-initialization, if a channel is paused just as the last transfer in a block completes (just as the transfer counter reaches zero), none of the register reloads take place (count, source address, and destination address). When the same channel is restarted, the channel will not transfer anything due to the zero transfer count. This problem only occurs at block boundaries. Internal reference number C601258.

WORKAROUND: Do not pause across block boundary if the internal FIFO is to be used by other channels for high performance. For DMA transfers with auto-initialization, if a channel is paused with a zero transfer count, manually reload all registers before restarting the channel.

Problem 0.0.6 DMA: Stopped transfer reprogrammed doesn’t wait for sync

If any non-synchronized transfer (ex: Auto-init Transfer) is stopped, and then the same channel is programmed to do a Write Synchronized Transfer (ex: Split-mode transfer), the write transfer does not wait for the Sync event. Internal reference number C601261.

WORKAROUND: Perform a non-synchronized dummy transfer of one element to/from the same location before starting the synchronized transfer.

Problem 0.0.7 DMA freezes if post-increment/decrement across port boundary

For any DMA transfers with source/dest address post-increment/decrement, if the last element to be transferred is aligned on a port boundary, then the DMA may freeze before transferring this element. A port boundary is the address boundary between external memory and program memory, between external memory and the peripheral address space, or between program memory and the peripheral address space.

The following conditions cause DMA to freeze:

- For non-sync and frame-sync transfers: if a channel is paused after the second-to-last element is read, when the channel is then restarted with a request to the address at a port boundary the DMA will freeze.
- For split-mode transfers or read/write-sync transfers: the DMA will freeze while transferring the element aligned on the port boundary. A continuous burst transfer with post-increment/decrement source/dest address does not exhibit this problem. Internal reference number C601300.

WORKAROUND: Do not transfer to boundary addresses if the DMA source/dest address is post-incremented/decremented.

Problem 0.0.8 DMA paused during emulation halt

When running an auto-initialized transfer, the DMA write state machine is halted during an emulation halt regardless of the value of EMOD in the DMA Channel Primary Control Register. The read state machine functions properly in this case. The problem exists only at block boundaries. If EMOD=1, this problem is irrelevant since the DMA channel is expected to pause during an emulation halt. Internal reference number C601301.

WORKAROUND: There is no workaround for EMOD=0. Expect DMA transfers to pause when the emulator stops the processor.

Problem 0.0.9 DMA: RSYNC=10000b (DSPINT) doesn't wait for sync

If RSYNC in the DMA Channel Primary Control Register is set to Host-port host to DSP interrupt (DSPINT – 10000b), the DMA channel would do the read transfer without waiting for the sync event. There is not a problem if WSYNC is set to DSPINT. Internal reference number C601302.

WORKAROUND: Do not synchronized DMA reads to DSPINT. If a DMA read is desired during a Host-port host to DSP interrupt, set RSYNC in the Primary Control Register to one of the EXT_INT events instead (EXT_INT4 – EXT_INT7) and have the host trigger an interrupt on that pin rather then by writing to HPIC.

Problem 0.0.10 CPU: L-unit interprets some integer instructions as double precision floating point instructions

The floating point .L unit incorrectly interprets an integer instruction as a double precision floating point instruction. As a result, the .L unit fails to execute a floating-point instruction that follows. The following set of .L unit integer instructions may result in a subsequent .L unit floating point instruction failing to execute:

Integer .L unit Instruction	Interpreted by Floating Point .L unit as
CMPGT (all opfields)†	ADDDP / SUBDP
CMPGTU (all opfields)†	ADDDP / SUBDP
SAT	ADDDP / SUBDP

† Code can be written so that the CMPGT and CMPGTU opcodes are not obvious. CMPLT/CMPLTU and CMPGT/CMPGTU are pseudo operations of each other, in the case when the operands are incorrectly arranged. For example, for the piece of code below:

```
CMPLT  .L1x    B1,A0,A0      ; src1 should not use the cross path, pseudo-op will be substituted
CMPGT  .L1x    A1,8,A1       ; only src1 can be a constant, pseudo-op will be substituted
```

The assembler leaves the instructions above in the list file (.lst), but performs the following operations instead:

```
CMPGT  .L1x    A0,B1,A0      ; only src2 uses the cross path
CMPLT  .L1x    8,A1,A1       ; only src1 could be a constant
```

When determining which int/fp instruction scenarios will result in a floating point failure, treat the integer instruction as if it were the floating point instruction specified in the table above and refer to Table 6-15 in the *TMS320C62x/C67x CPU and Instruction Set Reference Guide*. When applying the rules of the hazard table, note that it is only possible for a subsequent same-unit floating point instruction to fail.

An example failing code sequence is:

```
       LDH     .D1T1   *+A7(2),A5
||     CMPGT   .L1     A4,A0,A4
||     SUB     .L2X    B0,A5,B5

       XOR     .S1     1,A4,A4
||     INTSP   .L2     B5,B4
||     INTSP   .L1     A5,A7     ; failing instruction
```

The failure occurs on INTSP.L1, because the .L1 FP unit is still busy executing the false ADDDP triggered by the CMPGT.L1 executed in the previous cycle. Internal reference number 4.

WORKAROUND: For any code sequences with an integer CMPGT/CMPGTU/SAT in the first execute packet and a floating point operation to the same .L unit in the second execute packet, ensure that if the integer instruction were treated as an ADDDP/SUBDP instruction, the second execute packet would not encounter any hazards as outlined in Table 6-15 of the *TMS320C62x/C67x CPU and Instruction Set Reference Guide*.

Problem 0.0.11 CPU: S-unit interprets some integer instructions as double precision floating point instructions

The .S unit instruction decode block incorrectly instructs the floating-point pipeline to perform a double precision floating-point operation as the result of an integer instruction. A subsequent floating-point instruction to the same .S unit then fails to execute. The following set of .S unit integer instructions may cause a subsequent .S unit floating-point instruction to fail to execute:

SHR (opfields 110110 or 110100)

CLR (opfields 11 or 111111)

EXT (constant form or register form)

If the SHR/CLR/EXT instruction is not followed by any other non-SHR/CLR/EXT integer instruction to the same .S unit, a subsequent floating-point instruction to the same .S unit may fail to execute. This applies even if more than one execute packet exists between the SHR/CLR/EXT instruction and the floating-point instruction.

An example failing code sequence is:

```
         ADD       .L2     4, B5, B5
|| [ A2] STW       .D2T2   B6,*B3
|| [ A1] SUB       .D1     A1, 1, A1
||       SHR       .S1     A6, 14, A8

         NOP       1

         ADD       .L2     B9, B4, B4
||       MPYU      .M1     A8, A3, A8

         ADDSP     .L1     A4, A5, A5
||       CMPLTSP   .S1     A5, A9, A2    ; failing instruction
||       LDW       .D1T1   *++A0,A9
||       SHR       .S2     B4, 14, B8
```

The failure occurs on CMPLTSP.S1, because the .S1 FP unit is still busy executing the false floating point instruction triggered by the SHR.L1 executed previously. Internal reference number 5.

WORKAROUND: Ensure that the above three forms of .S unit integer operations (SHR, CLR, EXT) are followed by any other .S unit integer operation BEFORE executing an .S unit floating-point operation on that particular .S unit. Note that a NOP instruction does not count as a non-SHR/CLR/EXT instruction.

Problem 0.0.12 CPU: MPYSP/MPYDP underflow failure

In some cases the floating point .M unit produces an incorrect destination result for MPYSP and MPYDP instructions which underflow.

If each of the following conditions is true, an MPYSP or MPYDP instruction may deliver an incorrect destination result:

(1) The expected result of an MPYSP or MPYDP instruction underflows.
(2) The expected destination result is +/-SFPN.

The .M unit incorrectly produces an exponent equal to Emax instead of the expected Emin. The fraction, sign, and UNDER status bits are correct. If the instruction underflows and should produce a destination result of +/-0 instead of +/-SFPN, then the result produced is correct. Internal reference number 8.

WORKAROUND: Do not use MPYSP or MPYDP for numbers that may generate an underflow.

Problem 0.0.13 CPU: DPSP underflow failure

In some cases the floating point .L unit produces an incorrect result for DPSP instructions which underflow. Internal reference number 6, 10.

CASE1: If each of the four following conditions is true, a DPSP instruction may deliver an incorrect destination result and incorrect INEX and UNDER status bit (in the Floating-Point Multiplier Configuration Register) results:

(1) the expected result underflows
(2) the intermediate result fraction is incremented due to rounding
(3) the pre-rounded intermediate result exponent is non-zero
(4) Rmode is not 01 (truncate)

An example code/data sequence that will generate this error is:

```
MVK     0x17373ff5, A8
MVKH    0x17373ff5, A8
MVK     0x2f35e46b, A9
MVKH    0x2f35e46b, A9
NOP
NOP
NOP
NOP
DPSP    .L1     A9:A8, A5       ;A5 = 39af2359 (should be 00000000)
NOP
```

CASE 2: If each of the four following conditions is true, a DPSP instruction will deliver an incorrect destination result and incorrect UNDER and OVER status bit results:

(1) The expected result underflows.
(2) The intermediate result fraction is incremented due to rounding.
(3) Rounding causes a carry out of the incremented intermediate result fraction.
(4) The intermediate result exponent (calculated as the source operand's biased exponent minus 0x380) has a value of '1111111x' (in binary).

The delivered result incorrectly reflects an overflow condition, and not an underflow condition as expected.

Example

Before instruction:	Round mode = 0 (round toward nearest even integer) A1:A0 = 07ffffff ff800000
Failing instruction:	DPSP A1:A0, A2
Incorrect result:	A2 = 0x7f800000 with OVER and INEX (Expected A2 = 0x00000000 with UNDER and INEX)

WORKAROUND: If conversion results may underflow, disable rounding mode by setting Rmode = 01 (truncate). Do not use DPSP if the above conditions are true.

Problem 0.0.14 CPU: DPTRUNC/DPINT overflow failure

In some cases the floating point .L unit produces an incorrect result for DPINT and DPTRUNC instructions which overflow. Internal reference number 9.

If each of the three following conditions is true then a DPINT or DPTRUNC instruction may deliver an incorrect destination result and incorrect INEX and OVER status bit results:

(1) The source operand has a negative sign.
(2) The source operand has a biased exponent equal to 1055 (0x41f) causing the expected result to overflow.
(3) The intermediate result fraction is not rounded (DPTRUNC always meets this condition).
(4) Rmode=10 (round up) for DPINT instruction. This is irrelevant for DPTRUNC instruction.

Example 1

Before instruction:	A1:A0 = 0xc1f232bf 7321a000
Failing instruction:	DPTRUNC .L1 A1:A0, A2
Incorrect result:	A2 = 0xdcd408ce (should be A2 = 0x80000000)

Example 2

Before instruction:	FADCR = 0x00000400
	A1:A0 = 0xc1f4775a 6d3fc000
Failing instruction:	DPINT .L1 A1:A0, A2
Incorrect result:	A2 = 0xb88a592d (should be A2 = 0x80000000)

WORKAROUND: Do not use DPINT or DPTRUNC if the above four conditions are true.

Problem 0.0.15 CPU: L-unit floating point instructions failed to execute after ADDDP/SUBDP re-execution

The floating point .L unit incorrectly interprets an integer instruction and as a result re-executes a preceding double precision floating point instruction, causing a subsequent floating point instruction to fail.

If the following sequence occurs:

(1) an ADDDP or SUBDP is executed on an .L unit.
(2) 2 execute packets later, any of the following integer .L unit instructions is executed in the SAME .L unit: AND, OR, LMBD, NORM, CMPLT, SADD, CMPEQ, or ABS. (See “instr X” in the example below.)
(3) 2 execute packets later after (2), a non-ADDDP/SUBDP floating point instruction is executed in the SAME .L unit. (See “failing instruction” in the example below.)

Then the final floating point instruction will fail to execute correctly. An example of this failure is shown below:

```
        SUBDP  .L1     A13:A12, A9:A8, A3:A2

        NOP

        AND    .L1     6, A8, A5          ; instr X
||      STW    .D1     A2, *A15++
||      MV     .S2X    A3, B3

        AND    .L1     A9, A12, A6        ; instr Y
||      STW    .D1     A4, *A15++
||      MV     .S2X    A5, B5

||      STW    .D2     B3, *B15++
        SPINT  .L1     A8, A4             ; failing instruction
```

In the above code sequence, instr X must be one of the previously defined eight integer .L unit instructions for the failure to occur, instr Y can be any .L unit instruction. Any number of X/Y instr pairs can exist between the SUBDP and SPINT. Internal reference number 7.

WORKAROUND: Insert an additional NOP between the SUBDP and instr X.